Inside AppleTalk
Vol 1

# 030-M004

# Inside
# Appletalk

# Contents

# Preface

AppleTalk has been under development for over two years starting late in the Fall of 1983. Inside AppleTalk was first published in May of 1984 and made available at that time to potential developers of AppleTalk products. Since that date, the protocols have been implemented and tested with care by Apple Computer and by third party developers. This real-life experience has led to several enhancements and refinements. It has, furthermore, revealed certain inaccuracies in the documentation. Several additional protocols have been added to the architecture.

For these reasons Inside AppleTalk has been under constant revision and correction for several months. Although a more carefully edited and revised version will be published by Apple at a later date, we consider it important to get the information contained herein into the hands of developers as soon as possible. We are pleased to make this early form of the revised manual available to developers at this time. Please communicate any errors, omissions, unclear descriptions, etc., to the authors at the following address (no phone calls please):

> Gursharan S. Sidhu,
> Apple Computer Inc., M/S 27-O,
> 20525 Mariani Avenue,
> Cupertino, CA. 95014.

Use this address only to send us comments on this document. Please do not write to this address or call us for technical consultations and questions regarding AppleTalk. Those issues should be addressed to Apple's Technical Support Group.

This document has been divided into two parts. The first provides detailed specifications of the various protocols constituting the AppleTalk protocol architecture. This is the primary source for specification and description of the protocols themselves. We have strived hard to be machine independent. Although occasional hints are provided on implementation issues, this is not a description of how these protocols are actually implemented on any computer system.

The second part contains the user's manuals of the two main AppleTalk development tools: Peek and Poke. These tools were first provided by us in the spring of 1984 and have undergone many modifications and enhancements since that time. The descriptions provided here refer to the latest available versions (version 3.0 for Peek, and version 3.1 for Poke). For your convenience, a diskette containing these tools is provided with this manual.

Although the changes to Inside AppleTalk are too numerous to list in detail, here is a brief discussion of the principal ones.

## 1. ALAP clarifications:

The description of this protocol in our original 1984 document was imprecise in some respects. This chapter has been carefully rewritten and the protocol details specified in a detailed algorithmic manner using a Pascal-like language. The role of the synch pulse has been discussed more carefully.

## 2. New Protocols:

Five new protocols have been added to AppleTalk and one protocol has been replaced. These are:

> PAP -- the Printer Access Protocol,
> ZIP -- the Zone Information Protocol,
> ASP -- the AppleTalk Session Protocol,
> EP -- the Echo Protocol,
> AFP -- the AppleTalk Filing Protocol.

Complete specifications of all of these except the last are included here. All protocols described in this manual have been fully implemented and tested. The AppleTalk Filing Protocol (version 1.0) is being implemented and tested at this time; a completely revised specification of this protocol will be made available as soon as this activity reaches completion.

It should also be noted that a proposal form document on an AppleTalk Data Stream Protocol is being written and will replace our early document on DSP (Data Stream Protocol), which was never implemented and is no longer a part of the AppleTalk protocol architecture. Readers are advised to discard the old chapter on DSP to avoid future confusion with the newly defined ADSP.

## 3. Internetting Issues:

Interconnection of several AppleTalk networks into a larger internet system has been one of the areas of most intense AppleTalk activity at Apple. Several changes and refinements have been made to DDP, NBP and RTMP to correctly handle certain issues related to the use of network numbers and zone names. The Zone Information Protocol has been completely designed, implemented and tested. It is important for developers to carefully review the relevant chapters and ensure that their implementations of these protocols conform to the specifications in this document. Failure to do so will, in all likelihood, lead to improper operation of their devices on AppleTalk internets. With AppleTalk internet routers and back-bone bridges appearing in the market place at this time, these issues are expected to rapidly assume major significance.

In the discussion of DDP, the issue of when two network numbers are equivalent, and how to use network numbers when sending or receiving has been specified in detail.

Although RTMP has remained essentially the same as in the May 1984 version of Inside AppleTalk, several issues of detail have been experimentally examined and specified here. More specifically:

> • a bridge's routing algorithm has been more clearly stated
> • all algorithms have been better defined (especially as related to non-seed bridges)
> • the aging of ABridge is now required in non-bridge nodes
> • a new mechanism has been added for non-bridge nodes to rapidly acquire their network number
> • a discussion of internet topologies is provided
> • mechanisms for the support for ZIP commands are discussed.

With respect to NBP, the following issues have been elucidated:

- the use of zone '*' in NBP calls
- the algorithm used by bridges for converting NBP BrRq's to zone-wide NBP LkUp's
- NBP implementations should not respond to their own lookup requests
- case insensitivity of entity names
- NBP confirmation returns socket number
- null zone field same as '*'.

## 4. ATP Clarifications:

Several subtle issues related to ATP have been examined and clarified herein. These include a detailed discussion of transaction ID wrap around and an algorithm for TID generation that eliminates wrap around problems. Other details of interest are:

- ATP calls to DDP should ignore errors
- ATP does not return an error if exactly-once mode is requested but not available
- STS bit usage
- EOM bit is not available for use by ATP clients
- Infinite retries and optional socket on Send-Request call.

## 5. Printer Access Protocol (PAP):

The specification of the printer access protocol had formerly been included in the Inside LaserWriter manual. However, this protocol is now used to communicate over AppleTalk with other devices such as the ImageWriter-II. Also, the earlier discussion of PAP mixed the protocol's specification with a description of the somewhat restricted implementation of the protocol on the Macintosh. This caused the false impression that PAP allowed only one connection at a particular time. For these reasons, this chapter has been completely rewritten, making it implementation independent and more complete.

## 6. Table of Different Reserved Fields, Sockets, Protocol Types, etc.:

An appendix listing the different reserved values of statically-assigned sockets, ALAP, protocol types, and DDP protocol types is included in the Appendix.

# I. AppleTalk - An Introduction

The AppleTalk Personal Network has been designed as a simple, inexpensive and flexible way to interconnect computers, peripheral devices and servers. AppleTalk can be used in three major configurations:

• As a stand-alone local-area network, AppleTalk allows workgroups to exchange information and share resources like printers, file servers, modems, and other peripherals. In this configuration, a stand-alone AppleTalk network can be used to interconnect up to 32 nodes (computers, devices and servers).

• Through the use of bridging devices, large numbers of AppleTalk networks, each with up to 32 nodes, can be connected into large, complex internetworks. The nodes connected to this internetwork obtain the same services as on a stand-alone AppleTalk. Through gateway devices, AppleTalks can be hooked into other local-area networks such as Ethernet™.

• As a peripheral bus, AppleTalk can connect a Macintosh to its dedicated attached devices.

To achieve this range of flexibility, Apple has designed highly reliable, inexpensive hardware, and a sophisticated set of communications protocols.

## Hardware Specifications

At the physical level, AppleTalk has a bus topology consisting of a linear trunk cable with intervening connection boxes to which devices attach via a short drop cable. Since end-user installation is assumed, assembled trunk cables are sold in standard 2, and 10 meter lengths with molded miniature DIN connectors. The trunk cables consist of shielded, twisted-pair cable. Devices are connected to the trunk cable with AppleTalk connection boxes. The connection box is a small plastic case containing a transformer, resistive and capacitive circuits for noise immunity, and two 3-pin miniature DIN connectors with terminating switches to a 100 ohm terminating resistor. Attached to this box is an 18-inch drop cable which terminates at the device with either a DB-9, DB-25 or miniature DIN connector plug. Rolls of trunk cable and miniature DIN connectors are available allowing users to build custom network cable runs of lengths other than the standard 2 and 10 meters.

For a complete hardware specification, refer to the <u>AppleTalk Electrical/Mechanical Specification</u> and the <u>AppleTalk Transformer Specification</u>, included as Appendices B and C at the end of this manual.

## AppleTalk Protocol Architecture

Underlying all use of AppleTalk is a specific set of rules, or communication protocols. Apple has developed a set of protocols that correspond to the various layers of the International Standards Organization (ISO) Open Systems Interconnection (OSI) reference model. Protocols at the ISO-OSI layers 1 through 5 (Physical, Data Link, Network, Transport , and Session) form the core of the AppleTalk protocol architecture.

While Apple recommends the use of these protocols, communication over AppleTalk is not dependent on their exclusive use. The AppleTalk architecture is designed to allow developers to add special functions and features for their particular applications. All protocols are layered, functionally distinct entities, allowing easy access to and addition of alternative protocols. A software developer could, for instance, leverage upon the physical and data link layers to build a different protocol architecture.

When designing alternative protocols within a layered architecture, it is important to avoid implementing layers as separate modules with interfaces based on high-overhead calls to the operating system. A layered architecture is modular only with respect to the clear and distinct functionality provided at each layer. The intelligent design of interfaces between layers can avoid problems such as unnecessary "buffer copying" when passing data between layers.

The general method used by the AppleTalk architecture for moving user information up or down through the protocol layers is known as data encapsulation/decapsulation. A unit of user information is enclosed by a layer-specific header and/or trailer as it moves through each layer, from the user application down to the Data Link layer. The corresponding protocol layers at the receiving end examine and remove the layer-specific protocol information, as the user data moves up through the layers to the receiving user application.

The protocols comprising the AppleTalk architecture are outlined below. Figure I-1 summarizes this architecture.


## Physical Layer

The electrical and mechanical characteristics of AppleTalk correspond to the Physical layer of the ISO-OSI reference model. The Physical layer performs the functions of bit encoding/decoding, synchronization, signal transmission/reception, and carrier sense. The layered nature of the AppleTalk architecture allows for the Physical layer to be replaced by another medium as long as these functions are provided, and the interface between the Physical and Data Link layers is maintained.

A detailed discussion of the functions preformed by the Physical layer is provided in the Electrical Specifications chapter.


## AppleTalk Link Access Protocol (ALAP)

The AppleTalk Link Access Protocol (ALAP) corresponds to the Data Link layer of the ISO-OSI reference model. ALAP, which must be common to all systems on the bus, provides "best effort" delivery of information between devices, also known as *nodes*. It provides the basic underlying service of packet transmission between the devices connected to a single AppleTalk network. ALAP manages the encapsulation of data in a *frame* and then provides access to the bus for transmission and reception of frames.


## Datagram Delivery Protocol (DDP)

The Datagram Delivery Protocol (DDP) is found at the next level of the architecture, corresponding to the Network layer of the ISO-OSI model. While the ALAP protocol provides delivery of packets over a single AppleTalk network, the Datagram Delivery Protocol extends this mechanism to include a group of interconnected AppleTalk networks.

known as an *internet*. DDP also introduces the notion of logical addressable entities within nodes, known as *sockets*. *Datagrams* are packets of data exchanged between sockets; sockets may therefore be visualized simply as the source and destination of datagrams.

## Routing Table Maintenance Protocol (RTMP)

The AppleTalk architecture includes several protocols at the level above DDP, corresponding to the Transport layer of the ISO-OSI model. These protocols add different levels or types of functionality to the underlying datagram delivery service. This Transport layer protocol allows bridges/internet routers to dynamically discover routes to the different AppleTalk networks in an internet. Non-bridge nodes use a subset of RTMP (known as the *RTMP stub*) to determine the number of the network to which they are connected, as well as the node ID of a bridge on their network.

## Name-Binding Protocol (NBP)

Name-Binding Protocol (NBP), a Session layer protocol, lets network users use character string names for socket clients and network services. The basic function of NBP is the translation of a character string name into the internet socket address of the corresponding socket client.

NBP also introduces the concept of a *zone* -- an arbitrary subset of networks in an internet where each network is in one and only one zone.

## AppleTalk Transaction Protocol (ATP)

The AppleTalk Transaction Protocol (ATP) adds a measure of reliability by providing a loss-free "transaction" service between sockets. This allows exchanges between two socket "clients", in which one client requests the other to perform a particular task and report the result.

## Zone Information Protocol (ZIP)

This protocol maintains an internet-wide mapping of networks to zone names. ZIP is used by Name-Binding Protocol to determine which networks belong to a given zone.

## Echo Protocol (EP)

This is a very simple protocol that allows any node to send a datagram to any other node on an AppleTalk internet and receive an "echoed" copy of that packet in return. It is useful for probing to confirm the existence of a particular node, or to make round-trip delay measurements.

## AppleTalk Data Stream Protocol (ADSP)

This is a connection-oriented protocol providing a reliable, full-duplex, byte stream service between any two sockets on an AppleTalk internet. It ensures in-sequence, duplicate-free delivery of data over its connections.

## Additional Protocols

The above mentioned protocols constitute the "core" AppleTalk protocols, upon which most higher-level services will be built. In additional, Apple has developed, or is developing various of these higher-level protocols, as summarized below:

> Printer Access Protocol (PAP) - utilizes ATP XO to create a stream-like service for talking with the Apple LaserWriter™ and other stream-based devices.
>
> PostScript™ - developed by Adobe Systems® of Palo Alto. Presents a resolution-independent standard method of describing graphical and textual data. Utilized in talking to the Apple LaserWriter™, and other graphical devices. Detailed in *Inside LaserWriter*, published by Apple Computer and the *Postscript Language Reference Manual*, published by Addison-Wesley.
>
> AppleTalk Session Protocol (ASP) - a general protocol, built on ATP, providing session establishment, maintenance and teardown, along with request sequencing.
>
> AppleTalk Filing Protocol (AFP) - a presentation level protocol for accessing remote file systems, built on ASP.

Detailed discussions of each of the AppleTalk protocol layers, with the exception of PostScript™, are provided in subsequent chapters.
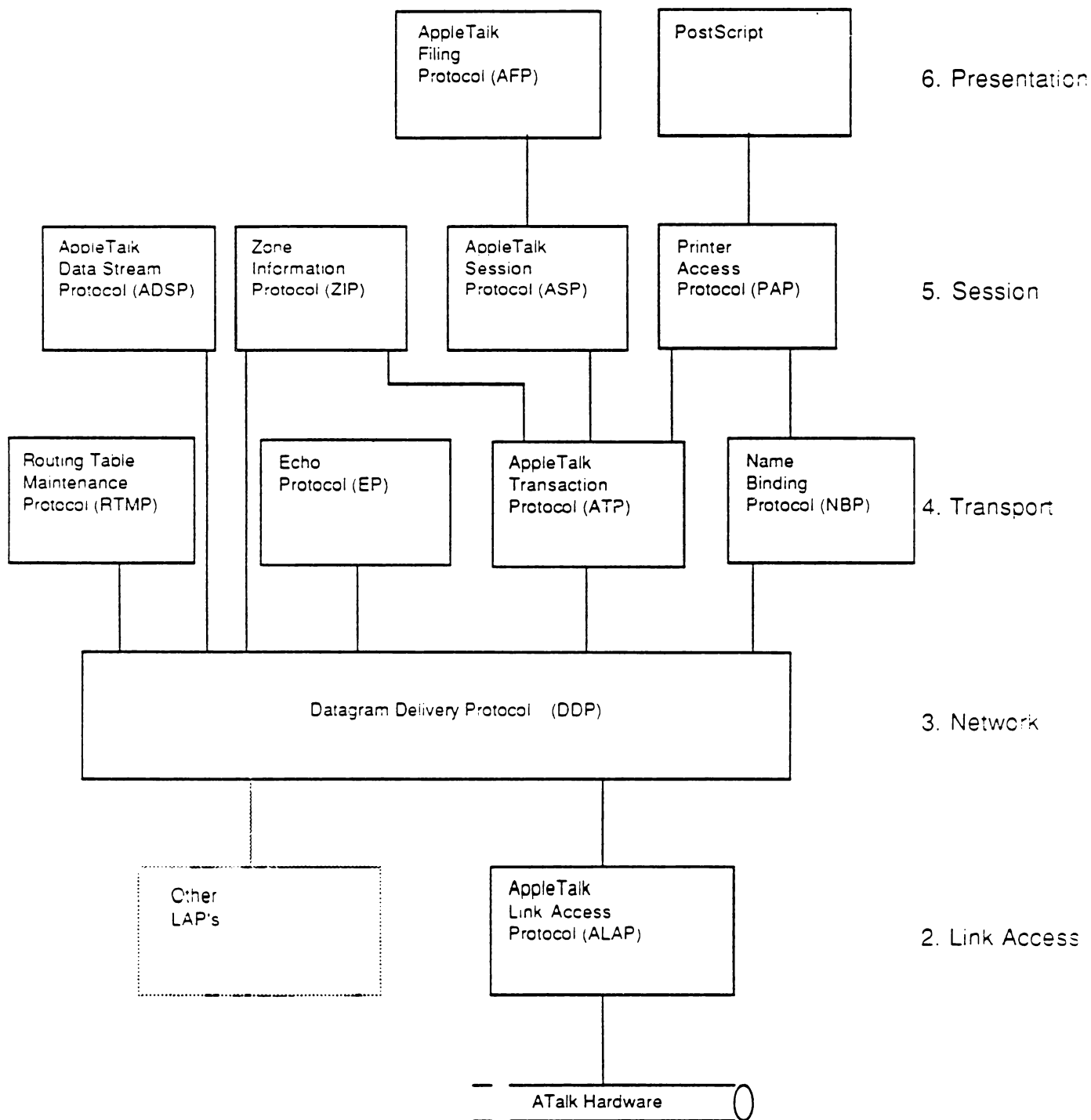
Figure I-1. AppleTalk Protocols: A Summary

# II. Electrical Specifications

The electrical and mechanical characteristics of AppleTalk correspond to the Physical layer of the ISO-OSI model. AppleTalk is a multi-drop, balanced, transformer-isolated serial communications system. The raw data rate is 230.4 kilobits per second over a maximum distance of 300 meters.

SDLC frame format is used, and FM-0 modulation (FM-0 is a bit-encoding technique that provides self-clocking). Balanced signalling is achieved using RS-422 drivers and receivers in each of the attached devices. The transformer provides ground isolation as well as protection from static discharge. Since devices are passively connected to the trunk cable via a drop cable, a device may fail without disturbing communications. Devices can be added and removed from the bus with only minor disruption of service.

The Physical layer performs the functions of bit encoding/decoding, signal transmission/reception, and carrier sense; these functions are discussed below.

### Bit Encoding and Decoding

Bits are encoded using a self-clocking technique known as FM-0 (bi-phase space). In FM-0, each bit cell (nominally, 4.34 microseconds) contains a transition at its end, thus providing timing information (known as one *bit-time*). Zeros (0's) are encoded by adding an additional transition at mid-cell (see figure II-1).

### Signal Transmission and Reception

The use of the EIA RS-422 signalling standard for transmission and reception over AppleTalk provides significantly higher data rates over longer distances than that of the EIA RS-232C standard. AppleTalk uses differential, balanced voltage signalling at 230.4 kilobits per second over a maximum distance of 300 meters. The balanced configuration provides better isolation from ground noise currents, and is not susceptible to fluctuating voltage potentials between system grounds or common-mode electromagnetic interference (EMI).

### Carrier Sense

The Physical layer provides an indication to ALAP when activity is sensed on the cable. Two indications are provided: SDLC frame in progress (the *hunt bit*), and missing clock detected. The hunt bit indicates when the hardware is searching for the start of the next SDLC frame -- when this bit is cleared, the hardware is in the middle of an SDLC frame. Note that a frame can not be detected until a complete flag has been sent on the line and recognized by the hardware. Missing clock, on the other hand, works in conjunction with the synchronization pulse sent before certain frames (see chapter III), to provide a more immediate indication of an ongoing transmission. Technically, missing clock indicates the detection, and then the absence, of a clocking signal on the line.
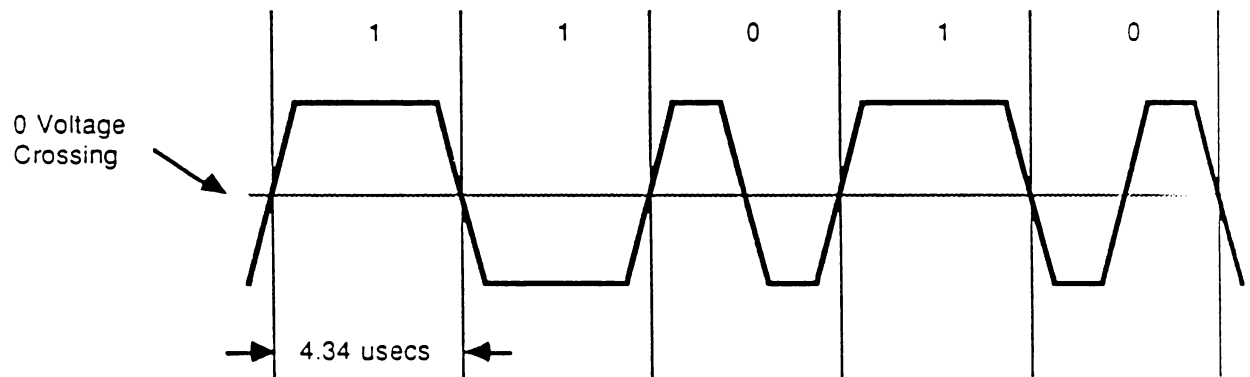
Figure II-1. FM-0 Encoding

# III.  AppleTalk Link Access Protocol (ALAP)

## About AppleTalk Link Access Protocol

The AppleTalk Link Access Protocol (ALAP), which corresponds to the data link layer of the ISO-OSI reference model, is the key mechanism allowing AppleTalk devices to share the communication medium (in this chapter also referred to as the *bus*).  ALAP, which must be common to all systems/devices on the bus, provides "best effort" delivery of information between these systems/devices, also known as *nodes*.  It provides the basic underlying service of packet transmission between the nodes connected to a single AppleTalk network.  ALAP manages the encapsulation of data in a *frame* and then provides access to the bus for transmission and reception of frames.  The main goals of the data link protocol for a shared-bus system like AppleTalk are as follows:

- provide bus access management
- provide a node addressing mechanism
- perform data transmission and reception
- ensure packet length and integrity

## Bus Access Management

All AppleTalk nodes compete for use of the bus.  It is the function of ALAP to resolve this contention and provide fair access for all nodes.  ALAP uses an access protocol known as *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA).  *Carrier sense* means that a node sending data first "senses" the line, and defers to any ongoing transmission.  *Collision avoidance* means that the protocol attempts to minimize collisions.  A collision occurs when two or more nodes transmit frames at the same time.  In the AppleTalk CSMA/CA technique, all transmitters wait until the line has been idle for a minimum time, plus an additional time as determined by the generation of pseudo-random numbers whose range is adjusted according to the perceived bus traffic.  AppleTalk hardware does not allow the actual detection of collisions.

## Node Addressing Mechanism

ALAP uses an 8-bit *node identifier number* (also known as the *node ID*) for identifying nodes on the bus.  ALAP is ultimately responsible for encoding both the destination and source node IDs in the header portion of an outgoing frame.  The destination node ID is used to "filter" frames at the data link layer; frames whose destination node ID does not match the receiving node's ID, or the broadcast ID ($FF) are not accepted at that node.

Unlike other networks that use fixed, universally-unique addresses, AppleTalk uses a dynamic node address assignment scheme.

### Dynamic Node ID Assignment

A key motivation for dynamic node ID assignment is that when a node is moved between networks, its old node ID should not conflict with a node ID already in use on the new network.  Furthermore, the availability of such a scheme eliminates one part of network configuration.  Unlike networks such as EtherNet no special universally-unique number

need be built into each node, nor is it necessary to administer the assignment of such numbers to different vendors.

When a node is activated on an AppleTalk network, it makes a guess at its own node ID, either by extracting this number from some form of long-term (e.g. parameter RAM or disk) memory, or by generating a random number (if the node has no non-volatile memory). The new node must then verify that this guessed number is not already in use on that network.

It does this by sending out a special frame, known as an ALAP *enquiry control frame*, to the guessed node address and waits for acknowledgement. If the guessed node ID is in use, then the corresponding node that is using it will, upon receiving the enquiry, respond with an ALAP *acknowledge control frame*. The reception of this frame tells the new node that the guessed node ID is already in use, and that the process must be repeated with a different guess. Each enquiry control frame is transmitted repeatedly to account for cases where a node already using the guessed node ID might be busy, and thus miss an enquiry frame.

Node ID numbers are divided into two classes: *server node IDs* and *user node IDs*. User node IDs are in the range 1 to 127 and server node IDs are in the range 128 to 254. A destination node ID of 255 has a special meaning. Frames received with the destination node ID equal to this value are accepted by all nodes; this permits the "broadcasting" of packets to all nodes on the network. A destination node ID equal to 0 is not allowed, and is treated as "unknown."

This division of node IDs is due to the fact that some nodes may, for extended periods of time, disable reception from AppleTalk (for instance, if they are engaged in a device-intensive operation such as disk access or transferring a bitmap document to a laser printer). Such a node would not respond to another node's enquiry frames, which could result in two node's acquiring the same node ID.

It is extremely important that no node acquire the same address as an already-functioning server node; this would disrupt service not only for the conflicting nodes, but for other users of the server as well. Excluding user (i.e non-server) node IDs from the server node ID range eliminates the possibility of user nodes (which are switched on and off with greater frequency) conflicting with servers.

Within the user node ID range, verification can be done more expeditiously (i.e. fewer retransmissions of the enquiry frame) to decrease the initialization time for such nodes. A more thorough node ID assignment scheme is used by servers (i.e. additional time taken, once chosen, to ensure that they will be unique on the network). This is not detrimental to the server's operation since such nodes are rarely switched on and off.

If during normal operation, after acquiring its node ID, a node detects the use of the same ID by some other node (by receiving a Clear-to-Send control frame with a source node ID identical to its own ID), ALAP should set a flag informing its client of this conflict condition. Recovery from this condition, if any, is the responsibility of higher-level protocols.

## Data Transmission and Reception

AppleTalk uses a bit-oriented data link protocol for data transmission and reception. Unlike byte-oriented protocols, a bit-oriented protocol permits the use of all possible bit

patterns within the frame. The frame delimiter for ALAP, known as a *flag* byte, is the distinguished bit sequence 01111110 ($7E). Typically, flags are generated by (hardware) transmitters at the beginning and end of frames; they are used by (hardware) receivers to detect frame boundaries.

In order for the data link protocol to transmit all possible bit patterns within a frame, the protocol must insure *data transparency*. ALAP does this with a technique known as "bit stuffing". When transmitting a frame, after each string of five consecutive 1's detected in the user data stream, it inserts a 0; this guarantees that data sent on the bus contains no sequences of more that five consecutive 1's. A receiving ALAP performs the inverse operation, "stripping" a 0 that follows five consecutive 1's.

Prior to transmitting a frame, ALAP sends out a synchronization pulse followed by a *frame preamble* consisting of two or more flags bytes. The frame is followed at its end by a *frame postamble/trailer* consisting of a flag byte and an abort sequence (seven or more 1 bits).

## ALAP Frame Format

An ALAP frame consists of a three-byte ALAP header followed by a variable length data field (0 to 600 bytes), and a 16-bit frame check sequence. Its format is summarized in figure III-1.

### ALAP Header

The ALAP header contains the destination node ID, the source node ID, and a one-byte *ALAP type field*. The ALAP type field specifies the type of frame. Values in the range 128 to 255 ($80 to $FF) are reserved for *ALAP control frames*; these frames do not contain a data field. The currently used ALAP control frames are as follows:

- *lapENQ* (ALAP type $81): an enquiry control frame, used in dynamic node ID assignment.

- *lapACK* (ALAP type $82): an acknowledgement control frame, sent in response to a *lapENQ*.

- *lapRTS* (ALAP type $84): a Request-to-Send control frame that notifies the destination node that the transmitter wants to send a data packet to it.

- *lapCTS* (ALAP type $85): a Clear-to-Send control frame sent in response to a *lapRTS*, indicating the willingness of the receiver to receive a data packet.

ALAP control frames received with values in the ALAP type field other than those listed above are simply discarded.

ALAP type fields with values in the range 1 to 127 ($01 to $7F) are used for *ALAP data frames;* these frames carry client data in the data field. In such frames, the type field is used to specify the ALAP protocol type of the client to whom the frames's data must be delivered. This allows the concurrent use of ALAP by several network layer protocols and is crucial to maintaining an "open systems architecture". The ALAP implementation in the receiving node uses the type field to determine the client for whom the data is intended. This client in turn uses this field to decide how to interpret the data (format of the data,

header for a higher-level protocol, etc.). As an example, the Datagram Delivery Protocol uses the values 1 and 2 in the ALAP type field.

ALAP simply transmits and receives data packets on behalf of its clients. The format and interpretation of the data field is defined by higher-level protocols, with the following exception: the low-order ten bits of the first two bytes of the data field must contain the length in bytes of the data field, more significant bits first (the data length includes the length field itself).

Frame Check Sequence Field

The 16-bit frame check sequence (FCS) is computed as a function of the contents of the destination node ID, source node ID, LAP type, and data fields, using the standard CRC-CCITT cyclic redundancy check algorithm. This is described in detail in the Algorithm section of this chapter.

Frame Size Limitations

Since the ALAP header is 3 bytes and the data field can contain from 0 to 600 bytes, the smallest valid frame (not including the preamble, postamble, and the FCS) is 3 bytes long, while the largest is 603 bytes.

## Frame Transmission

The transmission of a data frame by ALAP involves a special *dialogue* consisting of one or more ALAP control frames followed by the data frame. This dialogue is based on a CSMA/CA access protocol some aspects of which were outlined above in the section "Bus Access Management." The exact form of the dialogue depends on the destination of the frame; on this basis, ALAP distinguishes two kinds of frames: *directed* and *broadcast*. A directed packet is one whose destination address is a single node; a broadcast packet is intended to be received by all nodes.

Dialogues must be separated by a minimum *Inter-Dialogue Gap* (IDG) of 400 microseconds; the different frames of a single dialogue must follow one another with a maximum *Inter-Frame Gap* (IFG) of 200 microseconds.

The frame transmission procedure is described separately for directed and for broadcast frames. Figure III-2 diagrams the packets and timing used in this procedure.

Directed Data Frames

The transmitting node uses the physical layer's ability to sense if the line is in use. If the line is busy the node waits until it becomes idle following the current ongoing transmission. The node is said to *defer*. Upon sensing an idle line, the transmitter waits for a time equal to the minimum IDG plus a randomly generated amount. During this "wait," the transmitter continues to monitor the line. If the line becomes busy at any time during this wait period, the node must again defer. If the line remains idle throughout this wait period, then the node sends a *lapRTS* frame to the intended receiver of the data frame. The receiver must, within the maximum IFG, return a *lapCTS* frame to the transmitting node. Upon receiving this frame, the transmitter must within the maximum IFG send out the data frame.

The purpose of this algorithm is twofold: (1) to restrict the periods in which collisions are highly likely (this is during the *lapRTS-lapCTS* exchange), and (2) to spread out in time several transmitters waiting for the line to become idle. The *lapRTS-lapCTS* exchange, if successfully completed, signifies that a collision did not occur, and that all intending transmitters have heard of the coming data frame transmission and are deferring/waiting.

If in fact a collision does occur during the *lapRTS-lapCTS* exchange, a *lapCTS* will not be received, and the sending node will then *back off* and retry. The sending node is said to presume a collision.

The range of the random wait time generated by a node is adjusted if it presumes such a collision. The idea is that if collisions have been presumed for recently sent packets, this signifies heavier bus loading and higher contention for the bus. In this case, the random wait should be generated over a larger range, thus spreading out (in time) the different contenders for the line. Conversely, if the node has not had to defer on recent transmissions, a lighter load is signified and the random wait should be generated over a smaller range, thus reducing transmission time.

Two factors are used for adjusting the range: (a) the number of times the node had to defer, and (b) the number of times it presumed a collision. This history is maintained in two 8-bit *history bytes*, one each for deferences and collisions. At each attempt to send a packet these bytes are shifted left one bit. The lowest bit of each byte is then set if the node had to defer or presume a collision, respectively, on that attempt, else this bit is cleared. In effect, the history bytes remember the deference and collision history for the last eight attempts. The exact use of these bytes for determining the random wait times is described in the Algorithm section.

## Synchronization

At the beginning of all RTS frames, ALAP transmits a *synchronization pulse*. This is a transition on the bus, followed by an idle period greater than two bit-times. The synchronization pulse is obtained by momentarily enabling the line driver for at least one bit-time, before disabling it. This causes a transition that will be taken as a clock by all receivers on the network. However, since it is followed by an idle period of sufficient length, a *missing clock* is detected by the receivers. The missing clock allows transmitters to synchronize their access to the line (they become immediately aware if someone is about to transmit). Sync pulses may also be optionally sent at the beginning of other LAP frames.

## Directed Transmissions

Directed Packets are sent via a 3-frame dialogue as follows:

(1) The transmitter senses the bus until the bus is idle for the minimum IDG time.
(2) The transmitter waits an additional time determined by a random number.
(3) The transmitter sends a *lapRTS* frame to the intended destination node.
(4) The receiver sends a *lapCTS* frame to the transmitter.
(5) The transmitter, upon successful reception of the *lapCTS*, sends a data frame (in which it encapulates the client's packet).

Note: all response frames must be sent within the maximum IFG time.

The extra wait during step 2 tends to spread out transmitters which are contending for the line and, thus, minimize (or avoid) collisions. A transmitter presumes that a collision has occured when it does not receive a CTS frame in the required time (IFG). When this happens, the transmitter retries starting at step 1. For each attempt, a new random number must be generated in step 2. If after 32 attempts the transmitter is unable to send the data frame, it reports failure to its client.

<u>Broadcast Transmissions</u>

Broadcast packets are distinguished by their destination node ID being 255 ($FF). When ALAP wishes to send a broadcast packet, it performs the following procedure:

   (1) The transmitter senses the bus until the bus is idle for the minimum IDG time.
   (2) The transmitter waits an additional time determined by a random number.
   (3) The transmitter sends a *lapRTS* frame with destination address $FF.
   (4) The transmitter senses the line for the maximum IFG time.
   (5) After sensing the line idle for IFG, it sends its DATA frame.

The purpose of sending the *lapRTS* in step 3 is to ensure that other transmitters are cognizant of the intent to transmit, and to force a collision if another transmitter starts at the same time (that transmitter will then backoff). If the transmitter detects bus activity during step 4, it makes up to 32 further attempts beginning with step 1. If it fails to transmit the data frame after 32 attempts, it reports failure to its client.

Broadcast packets are sent without collision except in the unlikely case of another transmitter attempting a broadcast at the same time.

# Frame Reception

Frame reception is in many ways the inverse of the frame transmission process: frames are decapsulated and accepted only if their destination address is equal to the node's (or is equal to the broadcast address of 255) and their CRC is verified. The received frame can generate several conditions that must be handled by ALAP: overrun error, bad frame size, underrun error, bad frame type, and bad frame CRC.

<u>Frame Size Error</u>

If a frame (excluding the FCS), of greater than 603 bytes or smaller than 3 bytes is received then it is rejected and a bad frame error status generated.

<u>Overrun/Underrun Error</u>

If ALAP was not able to stay synchronized with the incoming data, causing receiver overruns or underruns, the frame will be rejected and an overrun or underrun error generated.

<u>Frame Type</u>

The frame's ALAP type field is checked against each of the valid values for an ALAP frame. If none match, the frame is rejected, and a bad frame-type error is generated.

Frame Check Sequence

The CRC-CCITT frame check sequence is computed for the incoming frame. When the entire frame has been received, ALAP tests the computed FCS for the frame. If the CRC is in error the frame is rejected and a bad frame-CRC error is generated.

## Algorithms

CRC-CCITT Calculation

The CRC-CCITT calculation algorithm uses the standard CRC-CCITT polynomial:

$$G(x) = x^{16} + x^{12} + x^5 + 1$$

The CRC-CCITT frame check sequence value corresponding to a given frame is calculated based on the following polynomial division identity:

$$\frac{M(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$

where:

M(x) = binary polynomial (corresponding to the frame after complementing its first 16 bits)

R(x) = remainder after dividing M(x) by the generating polynomial (its coefficients are the bits of the CRC)

Q(x) = quotient after dividing M(x) by the generating polynomial (its coefficients are ignored when calculating the CRC)

The implementation of the CRC for the FCS field, at the transmitter, computes the CRC starting with the first bit of the destination node ID following the opening flag and stopping at the end of the data field. The FCS field is equal to the ones-complement of the transmitter's remainder. The same calculation is performed at the receiver, however the FCS bytes themselves are included in the computation. The result of a correctly received transmission is then the binary constant 0001110100001111 ($x^{15}...x^0$). This constant is a characteristic of the divisor.

In addition to the division of the binary value of the data by the generating polynomial to generate the remainder for checking, the following manipulations occur:

• The dividend is initially preset to all ones before the computation begins; this has the effect of complementing the first 16 bits of the data.

• The transmitter's remainder is inverted bit-by-bit (FCS field) as it is sent to the receiver. The high-order bit of the FCS field is transmitted first ($x^{15}...x^0$).

If the receiver computation does not yield the binary constant 0001110100001111, the frame is discarded.

Random Wait Time Determination

The range of the random wait before attempting to send a frame after the line becomes idle is a function of the past history of attempted transmissions. As described in the "Frame Transmission" section, two history bytes are maintained indicating the number of presumed collisions and the number of deferrals in the last eight transmission attempts. These two bytes are used to adjust a *global backoff mask*. The mask takes on particular values between $00 and $0F (specifically binary 0, 1, 11, 111 and 1111). These values determine the range of random number picked, as specified below. The global backoff mask is adjusted at the beginning of a request to transmit a client's data as follows:

- The mask starts as 0

- If the number of times the node backed off during the last 8 attempts is greater than 2, the mask is extended by one bit up to the maximum of 4 bits (logical shift left and set the low-order bit) and the backoff history byte is then cleared.

- Else, if the number of times the node had to defer is less than 2, the mask is reduced by one bit (logical shift right) and the defer history byte is set to all one's.

- Else, if neither of these apply, the mask is left as is.


Due to collisions and deferrals, ALAP may have to make many attempts to send a packet. The following operations are performed, in order, before making the first attempt:

- The global backoff mask is adjusted as specified above.

- The two history bytes are shifted left one bit and the low-order bit of each is set to zero.

- The global backoff mask is copied into a *local backoff mask*.

During each attempt to send a packet, the following operations are performed, in order:

- If the line is busy, the node defers until the end of the dialogue. The low-order bit of the deferral history byte is set to one, and the low-order bit of the local backoff mask is also set to one (in case the mask was zero).

- If the line is not busy, the node waits 400 microseconds. If the line becomes busy during this time, the node defers as above.

- The random wait time is generated: a random number is picked and masked by (ANDed with) the local backoff mask. This random number thus has a range of zero to fifteen. The node waits for 100 microseconds multiplied by this random number. If the line becomes busy during this time, the node defers.

- The node sends a *lapRTS*. If a *lapCTS* is received within the maximum IFG (or if the packet is to be broadcast), the data is sent. If not (a collision is presumed), the low-order bit of the collision history byte is set to one, and the local backoff mask is shifted left one bit and its low-order bit set. The node then tries again.

- If, during an attempt to send a packet, 32 collisions occur or the node has to defer 32 times, the attempt is aborted and an error returned to the ALAP client.

## ALAP Procedural Model

The following procedural model, written in a Pascal-like language, is provided as a specification of ALAP. Any particular ALAP implementation's logical and timing behavior should follow this model.

### Assumptions

The model assumes that the program executes sufficiently fast so as to not introduce any execution delay into the timing of events. Where the ALAP specifies a timing delay, this is assumed to be performed via references to a global real function RealTime which returns the current time in microseconds. All timing constraints are specified as real constants.

The model assumes sequential, single-process execution of the code. This is especially true of TransmitPacket and ReceivePacket (and the procedures they call).

In a typical implementation, packet reception will be triggered by means of a hardware interrupt. The interrupt routine will then execute the ReceiveLinkMgmt procedure. The interrupt routine must provide a mechanism for saving valid data packets and for informing higher level protocols of this event. Such specifications are outside the scope of this document.

It is also assumed that a transmitter continuously listens to the bus while it is waiting for its access to the line.

### Global Constants, Types and Variables

The following global constants, types and variables are used throughout the model.

```
CONST
     minFrameSize = 3;              { smallest (LAP header only) frame }
     maxFrameSize = 605;           { size of largest LAP frame including FCS }
     maxDataSize = 600;            { size of largest (encapsulated) ALAP data field }
     bitTime = 4.34;               { bit time (µsec) }
     byteTime = 39.0;              { worst case single byte time (µsec) }
     minIDGtime = 400.0;           { minimum Inter-Dialog Gap (µsec) }
     IDGslottime = 100.0;          { slot time of transmit backoff algorithm (µsec) }
     maxIFGtime = 200.0;           { maximum Inter-Frame Gap (µsec) }
     maxDefers = 32;               { maximum defers for a single packet }
     maxCollsns = 32;              { maximum collisions for a single packet }
     lapENQ = $81;                 { LAP type field value of ENQuiry frame }
     lapACK = $82;                 {       ... ACKnowlodgement frame }
     lapRTS = $84;                 {       ... ReqeustToSend frame }
     lapCTS = $85;                 {       ... ClearToSend frame }
     hdlcFLAG = $7E;               { value of an HDLC FLAG }
     wksTries = 20;                { Number of ENQ sets for a workstation to try }

TYPE

{ global result types from LAP functions }
     TransmitStatus = (transmitOK, excessDefers, excessCollsns, dupAddress);
```

```
ReceiveStatus = (receiveOK, Receiving, nullReceive, frameError);
FrameStatus = (noFrame, lapDATAframe, lapENQframe, lapACKframe,lapRTSframe,
               lapCTSframe,badframeCRC,badframeSize, badframeType,
               overrunError, underrunError);
```

```
{ Data Link types and structures }
    bit = 0 .. 1;
    bitVector = PACKED ARRAY [0 .. 7] OF bit;
    octet = $00 .. $FF;
    anAddress = octet;
    aLAPtype = octet;
    aDataField = PACKED ARRAY [1 .. maxDataSzie] OF octet;
```

```
{ Basic structure of an ALAP frame, not including FLAGs, FCS }
    frameInterpretation = (raw, structured);
    aFrame = PACKED RECORD
             CASE frameInterpretation OF
                raw: rawData : PACKED ARRAY [1 .. maxFrameSize] OF octet;
                structured: (
                         dstAddr : anAddress;
                         srcAddr : anAddress;
                         lapType : aLAPtype;
                         dataField : a DataField)
             END;
```

```
VAR
    MyAddress : octet;           { set during initializeLAP }
    Backoff : INTEGER;           { current backoff range }
    fAdrValid,                   { MyAddress has been validated}
    FAdrInUse,                   { Another node has same MyAddress }
    fCTSexpected : BOOLEAN;      { RTS has been sent, CTS is valid }
    deferCount, collsnCount : INTEGER;    { optional, for statistics only }
    deferHistory, collsnHistory : bitVector;
    outgoingLength, incomingLength : INTEGER;
    outgoingPacket, incomingPacket : aFrame;
```

## Hardware Interface Declarations

The following declarations refer to hardware-specific interfaces which are assumed to be available to the LAP procedures. The functions are typically bits and/or bytes contained in the relevant hardware interface chip(s). Similarly, the procedures are expected to be represented in actual hardware by means of control bits within the hardware.

A brief description of the assumed attributes of each of these follows.

CarrierSense indicates that the hardware is sensing a frame on the bus
RcvDataAvail indicates that a data byte is available
rxDATA is the next data byte available (as indicated by RcvDataAvail)
EndOfFrame indicates that a valid closing FLAG has been detected
CRCok indicates that the received frame's FCS is correct (when EndOfFrame)
OverRun indicates that the code did not keep up with data reception
MissingClock indicates that a missing clock has been detected
setAddress sets the hardware to receive frames directed to MyAddress
enableTxDrivers and disableTxDrivers control the operation of the RS-422 drivers
enableTx and disableTx control the operation of the data transmitter
txFLAG causes the transmission of a FLAG

rxDATA causes the transmission of a data byte
txFCS causes the transmission of the Frame Check Sequence
txONEs causes 12+ one (1's) to be transmitted
resetRx, enableRx and disableRx control the receiver
resetMissingClock causes the MissingClock indication to be cleared

```
{ Hardware interface functions/procedures }
    FUNCTION CarrierSense : BOOLEAN;    EXTERNAL;
    FUNCTION RxDataAvail : BOOLEAN;     EXTERNAL;
    FUNCTION rxDATA : octet;            EXTERNAL;
    FUNCTION EndOfFrame : BOOLEAN;      EXTERNAL;
    FUNCTION CRCok : BOOLEAN;           EXTERNAL;
    FUNCTION OverRun : BOOLEAN;         EXTERNAL;
    FUNCTION MissingClock : BOOLEAN;    EXTERNAL;
    PROCEDURE setAddress (addr : octet);EXTERNAL;
    PROCEDURE enableTxDrivers;          EXTERNAL;
    PROCEDURE disableTxDrivers;         EXTERNAL;
    PROCEDURE enableTx;                 EXTERNAL;
    PROCEDURE txFLAG;                   EXTERNAL;
    PROCEDURE txDATA (data : octet);    EXTERNAL;
    PROCEDURE txFCS;                    EXTERNAL;
    PROCEDURE txONEs;                   EXTERNAL;
    PROCEDURE disableTx;                EXTERNAL;
    PROCEDURE resetRx;                  EXTERNAL;
    PROCEDURE enableRx;                 EXTERNAL;
    PROCEDURE disableRx;                EXTERNAL;
    PROCEDURE resetMissingClock;        EXTERNAL;
```

Interface Procedures and Functions

The ALAP model's interface to the next higher layer (its client) is specified in terms of the following three calls:

(1) PROCEDURE initializeLAP (hint : octet; server : BOOLEAN)

This procedure initializes the ALAP; it is expected to be called exactly one. The hint parameter is a suggested starting value for the node's AppleTalk physical link address: a value of zero indicates that ALAP should generate a starting value. Upon return from the call, the station's actual address is available in the global variable MyAddress. If server is true then the internal procedure acquireAddress will spend extra time to determine if another node is using the selected node address.

(2) FUNCTION transmitPacket (dstParam : anAddress; typeParam : aLAPtype;
        dataField : aDataField; dataLength : INTEGER) : TransmitStatus;

This is the call provided to transmit a packet. The internal function TransmitLinkMgmt performs the transmission Link Access algorithms.

(3) PROCEDURE receivePacket (VAR dstParam : anAddress;
        VAR srcParam : anAddress; VAR typeParam : aLAPtype;
        VAR dataField : aDataField; VAR dataLength : INTEGER);

This is the entry provided to receive a packet. The internal function ReceiveLinkMgmt implements the reception Link Access algorithms.


## InitializeLAP Procedure

initializeLAP is called to reset the global variables to known states; it calls acquireAddress to initialize MyAddress.

```
PROCEDURE initializeLAP (hint : octet; server : BOOLEAN);

VAR i : INTEGER;

BEGIN
    Backoff := 0;

    { intialize history data for Backoff calculations }
    FOR i := 0 TO 7 DO
      BEGIN
            deferHistory [i] := 0;
            collsnHistory[i] := 0;
      END;

    deferCount := 0;  collsnCount := 0;      { optional }

    acquireAddress (hint, server);
END;
```


## AcquireAddress Procedure

The procedure acquireAddress provides the dynamic node assignment algorithm. A special frame (of type lapENQ) is created and sent. When no node responds after repeated attempts, the current value of MyAddress is assumed to be safe for use by this node; the state of fAdrValid reflects this fact. If the global fAdrInUse ever becomes true after a call to AcquireAddress, another node that is using the same MyAddress has been detected.

```
PROCEDURE AcquireAddress (hint : octet; server : BOOLEAN);

VAR maxTrys, trys : INTEGER;  ENQframe : aFrame;

BEGIN
    IF hint > 0 THEN myAddress := hint
      ELSE IF server THEN MyAddress := Random (127) + 128
        ELSE MyAddress := Random (127) + 1;

    setAddress (MyAddress);
    fAdrValid := FALSE;
    IF server THEN maxTrys := wksTries * 6   { servers try six times as much as workstations }
      ELSE maxTrys := wksTries;

    trys := 0;  fAdrInUse := false;

    { the main loop of acquireAddress -- repeatedly check for response to ENQ }

    WHILE trys < maxTrys DO
```

```
         IF (transmitPacket (MyAddress, lapENQ, ENQframe.dataField,0) = transmitOK) OR
         fAdrInUse THEN BEGIN
             IF server THEN MyAddress := Random (127) + 128
              ELSE MyAddress := Random (127) + 1;
             setAddress (MyAddress);
             trys := 0;
         END          { IF }
         ELSE trys := trys + 1;
     END;   { WHILE }

     fAdrValid := TRUE;
  END;        { AcquireAddress }
```

## TransmitPacket Function

The function transmitPacket is called by the ALAP client to send a packet. After constructing (encapsulating) the caller's dataParam, it calls upon TransmitLinkMgmt to perform the actual link access.

```
FUNCTION transmitPacket (dstParam : anAddress; typeParam : aLAPtype;
                  dataParam : aDataField; dataLength : INTEGER) : TransmitStatus;

BEGIN
    IF fAdrInUse THEN transmitPacket := dupAddress
    ELSE BEGIN

        { copy interface data into frame for TransmitLinkMgmt }
        WITH outgoingPkt DO BEGIN
            dstAddr := dstParam;
            srcAddr := MyAddress;
            lapType := typeParam;
            dataField := dataParam;
        END; { WITH }

        outgoingLength := dataLength + 3;
        transmitPacket := TransmitLinkMgmt;
    END;   { ELSE }
END;        { transmitPacket }
```

## TransmitLinkMgmt Funtion

The function TransmitLinkMgmt implements the Carrier Sense, Multiple Access with Collision Avoidance algorithm; TransmitLinkMgmt is at the heart of the AppleTalk Link Access Protocol.

The typical AppleTalk hardware is not capable of performing collsions detection.

ALAP attempts to minimize collisions by requiring transmitters to wait a randomly generated amount of time before sending their RTS frames after the bus has been sensed idle for a minimum time (the IDG). Any transmitter which detects that another transmitter is in progress while it is in this random wait must defer.

In order to minimize delays under light loading, but still be able to minimize the probability of collsions under moderate to heavy loading, the random delay is picked in a range that is

constantly adjusted based upon the recently observed history of a node's attempts to access the bus. Two history vectors (deferHistory and collsnHistory) are used to keep track of the last 8 access attempts at the bus. Each attempt adds a single bit of history data to these vectors by shifting left the current values and updating the vacated bit with the appropriate value. DeferHistory[0] is set if a deferral is required during an access; collsnHistory[0] is set if a collsion is assumed.

The range of the current Backoff is adjusted upwards (to a maximum of 16) when the obeserved collsions exceeds 2 in the last 8 attempts; Backoff is adjusted downwards if the number of observed deferals is less than 2 in the last 8 attempts. When an adjustment is made, the corresponding history data is set to the "maximum" value so that further adjustments are inhibited until more history data has accumulated. The maximum value for deferHistory is all 1's; the maximum value for collsnHistory is all 0's.

Note that a local backoff (LclBackoff) is used during the retry attempts of a given frame. This has the effect of spreading out attempts to a non-listening node for a longer time, thus increasing its chances of receiving the packet.

```
    FUNCTION TransmitLinkMgmt : TransmitStatus;

VAR
    LclBackOff, i : INTEGER;
    fBroadcast, fENQ : BOOLEAN;
    xmttimer : REAL;
    rcvdframe : FrameStatus;
    RTSframe : aFrame;

BEGIN

WITH RTSframe DO BEGIN
    dstAddr := outgoingPacket.dstAddr;
    srcAddr := MyAddress;
    lapType := lapRTS
END;

fBroadcast := (outgoingPacket.dstAddr = $FF);
fENQ := (outgoingPacket.lapType = lapENQ);

{ Adjust Backoff, based upon recent history }
IF bitCount (collsnHistory) > 2 THEN BEGIN
    Backoff := min (max (Backoff * 2, 2), 16);
    FOR i := 0 TO 7 DO collsnHistory [i] := 0;
END         { IF }

ELSE IF bitCount (deferHistory) < 2 THEN BEGIN
    Backoff := Backoff DIV 2;
    FOR i := 0 TO 7 DO deferHistory[i] := 1
END;        { ELSE IF }

{ Shift history data }
FOR i := 7 DOWNTO 1 DO BEGIN
    collsnHistory [i] := collsnHistory [i-1];
    deferHistory [i] := deferHistory [i-1];
END;        { FOR }
collsnHistory [0] := 0; deferHistory [0] := 0;
```

```
{ Initialize main loop }
deferTries := 0;  collsnTries := 0;
LclBackoff := Backoff;  transmitdone := FALSE;

{ Begin main loop }
REPEAT

    { Wait for minimum Inter-Dialog Gap time }
    REPEAT

        { Wait for any packet in progress to pass }
        IF CarrierSense THEN BEGIN
                { Ensure minimum backoff if packet in progress }
                LclBackoff := max (LclBackoff, 2);
                deferHistory [0] := 1;

                { Perform watchdog reset of Rx for "stuck" CarrierSense }
                xmttimer := RealTime + 1.5 * maxFrameSize * byteTime;
                REPEAT UNTIL ( NOT CarrierSense) OR (RealTime > xmttimer);
                IF CarrierSense THEN ResetRx;          { something's wrong, clear it }
        END;  { IF }

        { Wait for minimum IDG after packet (or idle line) }
        xmttimer := RealTime + minIDGtime;
        REPEAT UNTIL (RealTime > xmttimer) OR CarrierSense;

    UNTIL NOT CarrierSense;

    ResetMissingClock;

    { Wait our additional backoff time, deferring to others }
    xmttimer := RealTime + Random (LclBackoff) * IDGslottime;
    REPEAT UNTIL (RealTime > xmttimer) OR CarrierSense;

    IF CarrierSense OR MissingClock THEN BEGIN   { defer }
        DeferCount := DeferCount + 1;          { optional }
        LclBackoff := max (LclBackoff, 2);
        deferHistory [0] := 1;
        IF deferTries < maxDefers THEN deferTries := deferTries + 1
        ELSE BEGIN
             TransmitLinkMgmt := excessDefers;
             transmitdone := TRUE;
        END    { ELSE }
    END    { IF }

    ELSE BEGIN    { NOT (CarrierSense OR MissingClock) }
       IF fENQ THEN transmitFrame (outgoingPacket, 3)
       ELSE transmitFrame (RTSFrame, 3);

       { use common code to detect line state }
       fCTSexpected := TRUE;
       rcvdframe := receiveFrame;
       fCTSexpected := FALSE;

       IF fAdrInUse THEN BEGIN
            TransmitLinkMgmt := dupAddress;
            transmitDone := TRUE;
       END    { IF }
```

```
          ELSE CASE rcvdFrame OF
            noFrame: IF fBroadcast THEN BEGIN
                transmitFrame (outgoingPacket, outgoingLength);
                transmitLinkMgmt := transmitOK;
                transmitdone := TRUE;
            END;

            lapCTSframe : IF (NOT fENQ) AND (NOT fBroadCast) THEN BEGIN
                transmitFrame (outgoingPacket, outgoingLength);
                transmitLinkMgmt := transmitOK;
                transmitdone := TRUE;
            END;
          END;           { CASE }

          { Assume collision if we don't receive the expected CTS }
          IF NOT transmitdone THEN BEGIN
            CollsnCount := CollsnCount + 1;              { optional }
            collsnHistory[0] := 1;                       { update history data }
            IF collsnTries < maxCollsns THEN BEGIN
                LclBackoff := min (max (LclBackoff*2,2),16);
                collsnTries := collsnTries + 1;
            END          { IF }
            ELSE BEGIN
                TransmitLinkMgmt := excessCollsns;
                transmitdone := true;
            END          { ELSE }
          END    { IF NOT ... }
       END    { ELSE NOT ... }

    UNTIL transmitdone;
    END;           { TransmitLinkMgmt }
```

## TransmitFrame Procedure

The procedure transmitFrame is responsible for putting data on the bus.  Notice that certain details, such as how a FLAG is forced and a packet terminated (which includes sending of the FCS) are not explicitly detailed here due to their extreme hardware dependence.  Note: the 12 ones at the end of the frame are required to finish clocking of the data by a receiver.

```
    PROCEDURE transmitFrame (VAR frame : aFrame; framesize : INTEGER);

VAR
    i : INTEGER;
    bittimer : REAL;

BEGIN

disableRx;

{ Generate the synchronizing pulse -- really only required before RTS frames}
bittimer := RealTimer + 1.5 * bitTime;
enableTxDrivers;
WHILE RealTime < bittimer DO BEGIN END;
disableTxDrivers;
bittimer := RealTimer + 1.5 * bitTimer;
```

```
WHILE RealTime < bittimer DO BEGIN END;

{ Start the actual frame transmission }
enableTxDrivers;
enableTx;
txFLAG;  txFLAG;              { Output 2 opening FLAG's }
FOR i := 1 to framesize DO TxData (frame.rawData[i]);
txFCS;                       { Send the FCS }
txFLAG;                      { Send the trailing FLAG }
txONEs;                      { Send 12 1's for extra clocks }
disableTxDrivers;

{ re-establish default listening mode }
resetMissingClock;
enableRx;

END;        { transmitFrame }
```

## ReceivePacket Procedure

The procedure receivePacket is the primary interface routine to higher levels. It is specified as if it is synchronously called by the user. Note that in many implementations, the lower-level ReceiveLinkMgmt function would be invoked by an interrupt routine.

```
PROCEDURE receivePacket ( VAR dstParam : anAddress; VAR srcParam : anAddress;
              VAR typeParam : aLAPtype; VAR dataParam : aDataField;
              VAR dataLength : INTEGER);

VAR status : ReceiveStatus;

BEGIN

REPEAT
    status := ReceiveLinkMgmt;
    IF status = receiveOK THEN BEGIN
      WITH incomingPacket DO BEGIN
        dstParam := dstAddr;
        srcParam := srcAddr;
        typeParam := lapType;
        dataParam := dataField;
      END;   { WITH }
      dataLength := incomingLength;
    END;     { IF }
UNTIL status = receiveOK

END;        { receivePacket }
```

## ReceiveLinkMgmt Function

The function ReceiveLinkMgmt implements the receiver side of the Link Access Protocol; it would typically be called from an interrupt routine rather than receivePacket.

```
FUNCTION ReceiveLinkMgmt : ReceiveStatus;

VAR
```

```
            status : ReceiveStatus;
            CTSframe, ACKframe : aFrame;

   BEGIN

   status := Receiving;
   WHILE status = Receiving DO
       CASE receiveFrame OF
           badframeCRC, badframeSize, badframeType, underrunError, overrunError:
                   status := frameError;

           lapENQframe : IF fAdrValid THEN BEGIN
                   ACKframe.dstAddr := incomingPacket.srcAddr;
                   ACKframe.srcAddr := MyAddress;
                   ACKframe.lapType := lapACK;
                   transmitFrame (ACKframe,3);
                   status := nullReceive;
               END         { IF }
               ELSE BEGIN
                   fAdrInUse := TRUE;
                   status := nullReceive;
               END;        { ELSE }

           lapRTSframe : IF fAdrValid THEN BEGIN
                   CTSframe.dstAddr := incomingPacket.srcAddr;
                   CTSframe.srcAddr := MyAddress;
                   CTSframe.lapType := lapCTS;
                   transmitFrame (CTSframe,3);
               END         { IF }
               ELSE BEGIN
                   fAdrInUse := TRUE;
                   status := nullReceive;
               END;        { ELSE }

           lapDATAframe : IF fAdrValid THEN status := receiveOK
               ELSE BEGIN
                   fAdrInUse := TRUE;
                   status := nullReceive;
               END;        { ELSE }

           noFrame: status := nullReceive;

       END;   { CASE }
   ReceiveLinkMgmt := status;

   END;       { ReceiveLinkMgmt }
```

## ReceiveFrame Function

The function receiveFrame is responsible for interacting with the hardware.

```
FUNCTION receiveFrame : FrameStatus;

VAR rcvtimer : REAL;

BEGIN
```

```
{ Provide timeout for idle line }
rcvtimer := RealTime + maxIDGtime;
REPEAT UNTIL CarrierSense OR (RealTime > rcvtimer);
IF NOT CarrierSense THEN BEGIN
    receiveFrame := noFrame;
    EXIT (receiveFrame);
END;        { IF }

{ Line is not idle, check if frame is for us }
rcvtimer := RealTime + maxIFGtime;
REPEAT UNTIL RxCharAvail OR (RealTime > rcvtimer);
IF RxCharAvail THEN BEGIN          { receive frame }
    error := FALSE;  framedone := FALSE;  incomingLength := 0;
    REPEAT
      rcvtimer := RealTime + 1.5 * byteTime;
      REPEAT UNTIL RxCharAvail OR (RealTime > rcvtimer);
      IF RxCharAvail THEN  BEGIN
        IF OverRun THEN BEGIN
            receiveFrame := overrunError;
            error := TRUE;
        END          { IF OverRun }

        ELSE IF incomingLength < maxFrameSize THEN BEGIN
            incomingLength := incomingLength + 1;
            incomingPacket.rawData [incomingLength] := rxDATA;
        END         { ELSE IF }

        ELSE BEGIN
            receiveFrame := badframeSize;
            error := TRUE;
        END;        { ELSE }

        IF EndOfFrame THEN
          IF CRCok THEN BEGIN
            incomingLength := incomingLength - 2;         { account for CRC }
            IF incomingLength < minFrameSize THEN BEGIN
              receiveFrame := badframeSize;
              error := TRUE;
            END    { IF incomingLength ... }
            ELSE framedone := TRUE;
          END       { IF CRCok }
          ELSE BEGIN       { bad CRC }
            receiveFrame := badframeCRC;
            error := TRUE;
          END;
      END            { IF RxCharAvail }

      ELSE BEGIN { RealTime > rcvtimer }
          receiveFrame := underrunError;
          error := TRUE;
      END    { ELSE }

    UNTIL framedone OR error;

    { Check on validity of the frame }
    IF framedone THEN
      IF fAdrValid THEN
          { if our address if valid, check on actual type }
```

```
IF incomingPacket.lapType < $80 THEN receiveFrame := lapDATAframe;
ELSE CASE incomingPacket.lapType OF
    lapENQ : receiveFrame := lapENQframe;

    lapACK : BEGIN
        receiveFrame := lapACKframe;
        fAdrInUse := TRUE;
      END; { lapACK }

    lapRTS : receiveFrame := lapRTSframe;

    lapCTS : IF fCTSexpected THEN receiveFrame := lapCTSframe
      ELSE BEGIN
        fAdrInUse := TRUE;
        receiveFrame := badframeType
      END;   { ELSE }

    OTHERWISE receiveFrame := badframeType;
  END      { CASE }
ELSE IF incomingPacket.rawData[1] <> $FF THEN BEGIN
    fAdrInUse := TRUE;     { We received something we didn't expect }
    receiveFrame := noFrame;
  END { ELSE IF }
  END    { IF RxCharAvail }
ELSE receiveFrame := noFrame;    { no CharAvail }

resetRx;  resetMissingClock

END;      { receiveFrame }
```

Miscellaneous Functions

The following low-level routines are referenced by the above code:

```
FUNCTION bitCount (bits : bitVector ) : INTEGER;
    VAR i, sum : INTEGER;
    BEGIN
      sum := 0;
      FOR i := 0 TO 7 DO sum := sum + bits [i];
      bitCount := sum
    END;  { bitCount }

FUNCTION min (val1, val2 : INTEGER) : INTEGER;
    BEGIN
      IF val1 < val2 THEN min := val1
      ELSE min := val2
    END;  { min }

FUNCTION max (val1, val2 : INTEGER) : INTEGER;
    BEGIN
      IF val1 > val2 THEN max:= val1
      ELSE max:= val2
    END;  { max }

FUNCTION Random (maxval : INTEGER) : INTEGER;
    BEGIN
            { this function is implemented as any "good" pseudo-random number generator
```

>                 which produces a result inthe range 0 .. maxval-1 }
>         END;


## SCC Implementation

One of the integrated circuits most commonly used in the implementation of ALAP is the Zilog 8530 Serial Communications Controller (SCC). This section explains how the hardware interface routines declared above could be implemented with that device. This discussion should not be construed as in any way implying that the SCC must be used in the implementation of ALAP. Many other devices can be employed to effectively implement ALAP. Note that all of the registers and bit names below are those used by Zilog in its SCC documentation.

CarrierSense indicates that the hardware is sensing a frame on the bus: this corresponds to the complement of the SYNC/HUNT bit in RR0.

RcvDataAvail indicates that a data byte is available; this corresponds to the Rx Character Available bit in RR0.

rxDATA is the next byte from the bus; it is RR8.

EndOfFrame indicates that a valid closing FLAG has been detected; this is the End of Frame bit in RR1.

CRCok indicates that the received frame's FCS was correct (when EndOfFrame is true); this is the complement of the CRC/Framing Error bit in RR1.

OverRun indicates that the code did not keep up with data reception; this is the Rx Overrun Error bit in RR1.

MissingClock indicates that the hardware has detected a missing transition on the bus: this is the One Clock Missing bit in RR10.

setAddress is a procedure which sets the hardware to receive packets whose destination address matches MyAddress; this simply sets WR6 in the SCC.

enableTxDrivers and disableTxDrivers control the operation of the RS-422 drivers. This would generally be controlled by one of the SCC's output bits (on the Macintosh, it's the RTS bit in WR5).

enableTx and disableTx control the operation of the transmitter; this is done by means of the Tx Enable bit in WR5.

txFlag causes the transmission of a FLAG. THis happens automatically at frame opening when Tx Enable is set; however code must delay long enough to cause the extra FLAG. The trailing FLAG is generated automatically at frame end as part of the Tx Underrun processing.

txDATA causes the transmission of a data byte; this is WR8.

txFCS causes the transmission of the Frame Check Sequence. This is caused automatically by letting Tx Underrun occur.

txONEs causes 12+ ones (1's). This is done by disabling the SCC transmitter (setting TX Enable to zero) while leaving the RS-422 drivers on and delaying.

resetRx, enableRx and disableRx control the receiver; this is done by means of the Rx Enable bit in WR3. resetRx should also flush the receive FIFO.

resetMissingClock causes the MissingClock indication to be cleared; this is done by a Reset Missing Clock command via WR14.

One Byte (8 bits)

MSB                                    LSB

Frame Preamble ─── Flag

Flag

Destination Address

Source Address

LAP Type

ALAP Frame ─── Length Field

Data Field
(up to 600 bytes)

Frame Check
Sequence

Frame Postamble ─── Flag

Abort
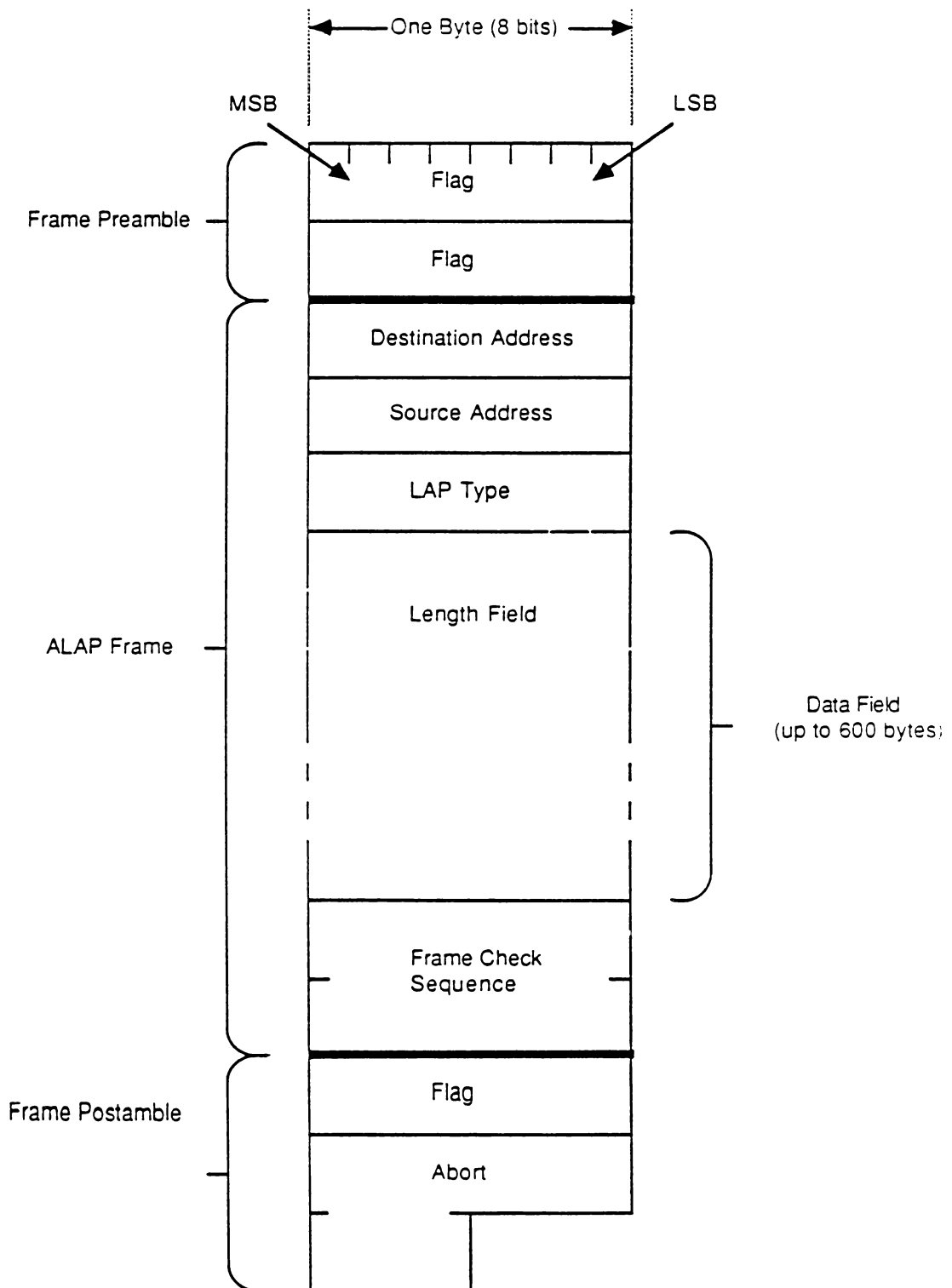
Figure III-1. ALAP Frame Format

End of previous frame
(Line becomes idle)

RTS          CTS          DATA

$T_D + T_R$          t          t

(a) Directed packet

Line is sensed idle,
retry initiated

RTS

RTS

Collision          200 μsec

(b) Collision

RTS          DATA

200 μsec

(c) Broadcast packet

Generated if another node with
same node ID is present

ENQ          ACK

t

(d) Node address assignment dialogue
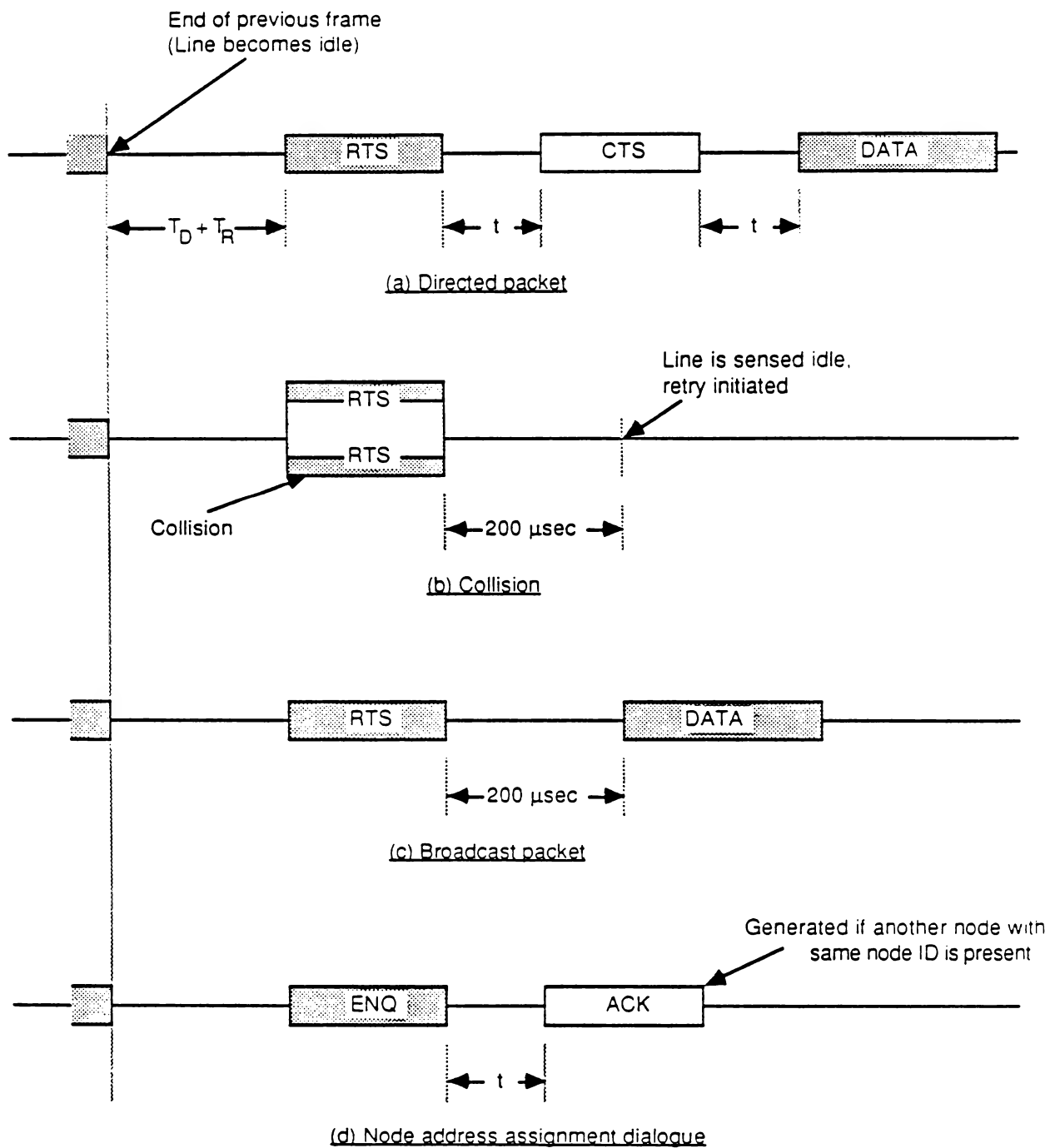
t  --  Inter-Frame Gap ( less than 200 μsecs)

$T_D$ --  Inter-Dialogue Gap (greater than 400 μsecs)

$T_R$ --  randomly generated time interval

Figure III-2. ALAP Timing Diagrams

# IV. Datagram Delivery Protocol (DDP)

## About Datagram Delivery Protocol

While ALAP provides a "best effort" node-to-node delivery of packets on a single AppleTalk network, the Datagram Delivery Protocol (DDP) is designed to extend this mechanism to the socket-to-socket delivery of datagrams over an AppleTalk internet (see figure IV-1). An AppleTalk internet consists of one or more AppleTalk networks connected by intelligent nodes referred to as *bridges* or *internet routers*. (Bridges should not be confused with *gateways*, which are nodes that separate and manage communication between different types of networks.)

Bridges are packet-forwarding agents. Packets can thus be sent between any two nodes of an internet by using a "store and forward" process through a series of internet routers. By combining this facility with DDP, packets can be sent between any two sockets on an internet.

A bridge will often consist of a single node connected to two AppleTalk networks, or it might consist of two nodes, each of which is connected to the other through a communication channel. As far as the protocol architecture is concerned, the channel between the two halves of a bridge could take a variety of forms: a leased or dial-up line, another network (e.g. a wide-area packet-switched or circuit-switched public network), or a higher-speed broadband or baseband local-area network used as a "backbone".

## Sockets and Socket Identification

*Sockets* are logical entities within the nodes connected to an AppleTalk internet. Sockets are owned by *socket clients*, which are typically processes (or functions in processes), implemented in software in the node. A socket client can send and receive *datagrams* (this term is used to signify packets of data carried by DDP between the sockets of an internet) only through sockets that it owns. Each socket within a given node is identified by an eight-bit socket number. Socket numbers are treated as unsigned integers. There can be at most 254 different socket numbers in a given node (the values 0 and 255 are reserved and can not be used to identify sockets).

Sockets are classified into two groups: *statically-assigned* and *dynamically-assigned*. Statically-assigned sockets (SAS) have socket numbers in the range 1 through 127; these sockets are reserved for use by clients such as the AppleTalk core protocols (e.g. ATP, NBP, RTMP, etc.) and for low-level built-in network services such as echoers. Socket numbers decimal 1 through 63 are specifically reserved for use by Apple. Socket numbers decimal 64 through 127 are available for unrestricted experimental use. Use of these experimental SASs is not recommended for commercial products, since there is no mechanism for eliminating conflicting usage of the same socket(s) by different developers (see the section "Use of Name-Binding Protocol with Sockets"). See Appendix A for a summary of socket number usage.

Socket numbers decimal 128 through 254 are assigned dynamically by DDP upon request from clients in that node; sockets of this type are said to be dynamically assigned (DAS).

No two sockets in the same node can have the same socket number. Thus, the socket number taken together with the ALAP node ID provides an unique identifier for any socket on a single AppleTalk network. This is called the socket's *AppleTalk address*.

A *network number* is a 16-bit number that uniquely identifies a network in an internet. By assigning a network number to each AppleTalk network in an internet, it is possible to uniquely identify any socket on the internet. The *internet address* of a socket consists of its socket number, the node ID (of the node in which the socket is located) and the network number (of the network on which the node is located). Thus the source and destination sockets of a datagram can be fully specified by their internet addresses.

The network number 0 is reserved to mean unknown; by default it specifies the local network to which the node is connected. Packets whose destination network number is zero are addressed to a node on the local network. This allows systems consisting of a single AppleTalk network to operate without the need for an explicit network number. The value 65,535 (all bits set to one) is reserved for future use. Although we do not expect our clients to build such a large internet, the two-byte network number theoretically allows proper operation on an internet with up to 65,534 AppleTalk networks.

## DDP Protocol Type Field

The AppleTalk architecture allows the implementation of a large number (up to 255) of parallel protocols that are clients of DDP (and hence are located above DDP). It is important to realize that socket numbers are not associated with a particular protocol type and should not be used to demultiplex among parallel protocols at the transport level. Instead, for this purpose a one-byte *DDP protocol type field* is provided in the DDP header. See Appendix A for a summary of the usage of the DDP protocol type field.

## Socket Listeners

Socket clients provide code, referred to as the *socket listener*, that "receives" datagrams addressed to that socket. The specific implementation of a socket listener is node-dependent. For efficiency, the socket listener should be able to receive datagrams asynchronously through either an interrupt mechanism or an I/O request completion routine.

The code that implements the DDP in the node must contain a data structure called a *sockets table* to maintain an appropriate descriptor of each active socket's listener.

## DDP Interface

As shown in figure IV-2, the DDP interface is the boundary at which the socket client can issue calls to and obtain responses from the DDP implementation module in the node. The DDP Implementation Module supports four calls, described below.

### Opening a Statically-Assigned Socket

The caller specifies the socket number and the socket listener for that socket. The call returns with a result code:

- success: socket activated

- error:    various cases: socket already active, not a statically-assigned socket
            (outside the permissible range), socket table full

## Opening a Dynamically-Assigned Socket

This is similar to the preceding except that the caller does not specify the socket number. The call returns a result code and, if successful, the activated socket's number. The result code takes the following possible values:

- success:  socket activated
- error:    various cases: socket table full, all dynamic sockets busy

## Closing a Socket

This request specifies the number of the socket to be deactivated. If the socket is currently active, it is removed from the sockets table. The result code has the following possible values:

- success:  socket deactivated
- error:    no such socket

## Sending a Datagram

The request specifies the number of the source socket, the internet address of the destination socket and the DDP protocol type field value. Also the length and location of the data part of the datagram are provided in the request. Since DDP includes an optional software checksum in internet datagrams, the requester must specify whether or not this checksum is to be generated. The result code has the following possible values:

- success:  datagram sent
- error:    sending socket not active or valid, datagram too long

In addition to these four calls, a socket listener must provide a mechanism for the reception of datagrams.

## Datagram Reception by the Socket Listener

We do not specify this function since it is dependent on the implementation in the node. Some mechanism is needed to deliver datagrams within the node to the destination socket's listener. The DDP package should attempt this only if the destination socket is currently active. *The DDP must discard datagrams addressed to inactive sockets.* Furthermore, internet datagrams received with an invalid DDP checksum must be discarded.

## DDP Internal Algorithm

DDP is a simple, best-effort protocol for delivery of datagrams. As such there is no mechanism for recovery from packet loss or error situations.

Within the DDP implementation module, the primary function is to form the DDP header on the basis of the destination address, and then to pass the packet on to the appropriate ALAP. Similarly, for packets received from ALAP, DDP must examine the datagram's destination address in the DDP header and route the datagram accordingly. Details of this

operation depend on whether the node is a bridge; this is discussed more fully in the section "DDP Routing Algorithm".

## DDP Packet Format

A datagram consists of the DDP header followed immediately by the data.

There is a 10-bit datagram length field in the DDP header (see figures IV-3 and IV-4). The value in this field is the length in bytes of the datagram starting with the first byte of the DDP header and including all bytes up to the last byte of the data part of the datagram. Upon receiving a datagram, the receiving node's DDP implementation must reject all datagrams whose indicated length is not equal to the actual received length. The maximum length of the data component of a datagram is 586 bytes; longer datagrams must be rejected.

### Short and Long Form Headers

In addition, the DDP header contains the source and destination socket addresses and the DDP protocol type. Each of these addresses could be specified as a four-byte internet address. However, for packets whose source and destination sockets are on the same network, the network number fields are unnecessary; likewise, for such datagrams the source and destination node IDs are already found in the ALAP header, and would thus be redundant in the DDP header. For these reasons, DDP uses two types of header: a short form and an extended (or long) form.

DDP uses the value of the ALAP protocol type field to determine if the packet has a short or an extended DDP header. The ALAP protocol type field value is 1 for the short form, and 2 for the extended form.

The short form version of a DDP datagram is shown in figure IV-3. The datagram header is five bytes long. The first two bytes of the header contain the datagram length, with the more significant byte first. The upper 6 bits of this byte are not significant and should be set to zero. The datagram length field is followed by a one-byte destination socket number, a one-byte source socket number, and a one-byte DDP protocol type field. Such short header datagrams are sent if the source and destination sockets are on the same network.

The extended form datagram is shown in figure IV-4. The extended form DDP header is thirteen bytes long. It contains the full internet addresses of the source and destination sockets, as well as the datagram length and DDP type fields. For such packets, there is a 6-bit *hop count* field (described below) in the most significant bits of the first byte of the DDP header. Datagrams exchanged between sockets on different AppleTalk networks of an internet must use this form of header. In addition, the extended header includes a two-byte (16-bit) DDP checksum field. If the sending client so desires, the source node's DDP implementation calculates and inserts a software-generated checksum into this field; otherwise, the sending node sets this field to 0. The datagram's destination node recomputes this checksum and rejects the datagram if the received and computed values do not agree. All two-byte fields are specified high byte first.

The DDP checksum has been provided to allow the detection of errors caused by faulty operation (e.g. memory and data bus errors) within bridges on the internet. Implementors of DDP should treat it as an optional feature.

<u>DDP Checksum Computation</u>

The 16-bit DDP checksum is computed as follows:

> CkSum := 0 ;
> FOR EACH datagram byte starting with the byte immediately following the CkSum
> REPEAT the following algorithm:
>> CkSum := CkSum + byte; (unsigned addition)
>> Rotate CkSum left one bit, rotating the most significant bit into the least significant bit;
> IF, at the end, CkSum = 0 THEN
>> CkSum := hex FFFF (all ones).

Reception of a datagram with CkSum = 0 implies that it is unchecksummed.

<u>Hop Counts</u>

For datagrams that are exchanged between sockets on two different AppleTalk networks in an internet, a provision is made to limit the maximum number of internet routers the datagram can visit. This is done by including, in such internet datagrams, a hop count field.

The source node of the datagram sets this field to zero before sending the datagram. Each internet router increments this field by one. A bridge receiving a datagram with a hop count value of 15 should not forward it to another bridge; if such a datagram's destination is on a local network directly connected to the bridge, then the bridge should send it to that destination, otherwise the datagram should be discarded by the bridge. This provision is made to filter out of the internet packets that might be circulating in closed routes. Such closed routes (loops) are a transient situation that can occur for short periods of time while the routing tables are being updated by the Routing Table Maintenance Protocol (RTMP). Non-bridge nodes ignore this field.

The upper two bits of the hop count currently are not used by DDP, but are reserved for future use (such as the extension of the maximum value of the hop count beyond the currently allowed value of 15).

## DDP Routing Algorithm

A datagram is conveyed from its source to its destination socket over the internet through the bridges. The DDP implementation in the source node examines the destination network number of the datagram and determines whether the destination is on the local network or not. If it is, the short form DDP header is adequate, and the ALAP layer is called to send the packet to its destination node. If, however, the destination is not on the local network, DDP builds the extended header and calls ALAP to send the packet to a bridge (if there is more than one such bridge, any one will do) on the local network. The bridges examine the destination network number of the datagram, and use routing tables to forward the datagram to subsequent bridges (through the ALAPs of appropriate intervening local networks to which the bridge is connected) to get it to a bridge connected to the destination network. There the datagram is sent to its destination node through the local network's link level protocol.

Each node on an AppleTalk maintains as an internally stored value the network number of the local network to which it is attached. The DDP implementation in a datagram's source

node determines if the destination network is the local network or not by comparing the destination network number to the internally stored value of the local network's number. If the two numbers are the same then a short DDP header is built for the datagram. An important special case arises when the internally stored value is zero (unknown). In this case, if the datagram's destination network number is non-zero, then DDP should assume that the packet is intended for a node on the local network. However, it must, in this case, build an extended DDP header and call ALAP to send the packet to the specified destination node *on the local net*. The extended DDP header enables the receiving node to throw away the packet if it determines it was not the intended recipient (the destination network of the packet is not equal to its internally stored local network number).

Routing tables are maintained by bridges by using the Routing Table Maintenance Protocol (RTMP) discussed in a separate chapter. The routing tables indicate, for each network number in the internet, the node ID (on the appropriate local network) of the next bridge on the proper route.

Non-bridge nodes do not need to maintain these tables. Such nodes need only two pieces of information: the network number of the local network (THIS-NET), and the node ID of any bridge (A-BRIDGE) on the local network. This can be done by implementing a simple subset of RTMP, called the *RTMP Stub*, in each such node. For nodes on systems consisting of a single network, the values of THIS-NET and A-BRIDGE will be zero ("unknown").

The internal routing algorithm of the DDP implementation module for a non-bridge node is given below. The sending client issues the send datagram call, specifying the destination socket's internet address.

```
IF (destination network number = 0)  OR  (destination network number = THIS-NET)  THEN
        BEGIN
                build the short-form DDP header;
                call ALAP to send it to the destination node
        END
ELSE
        BEGIN
                build the extended-form DDP header;
                IF THIS-NET = 0 THEN call ALAP to send the packet to the destination node
                ELSE IF A-BRIDGE <> 0 THEN call ALAP to send the packet to A-BRIDGE
                ELSE return an error (no bridge available)
        END;
```

For packets received from ALAP by non-bridge nodes, the routing function is simply one of delivering the packet to the destination socket in the node. DDP must first verify that the destination network number in an extended DDP header is in fact equivalent to that node's internally stored value of THIS-NET, and ignore the packet if not. (For a precise definition of this equivalence see the discussion at the end of this chapter.) It is also advisable for such nodes to verify that the destination node ID in an extended DDP header matches the station's node ID (or is equal to the broadcast address, $FF).

In internet routers, the routing algorithm is somewhat more complex; details are provided in the chapter on RTMP.

## Use of Name-Binding Protocol in Conjunction with Sockets

Developers of products for AppleTalk should not use statically assigned sockets except for purely experimental purposes. This restriction is imposed in order to avoid the conflicting use of the same statically assigned socket by different developers. Such conflicts are difficult to avoid in the absence of a central administering body.

Developers should instead use the name-binding technique to allow work stations to discover their server/service access point addresses. Thus developers would identify their server/service by a unique name. Workstations would then use NBP to bind an address to this name (for details, see the section on NBP). Once the client process has determined the proper destination socket number, it can then proceed to transmit packets to that socket.

Clearly, this requires that developers must implement NBP in their servers. While not significant for larger, complex servers, this could pose a significant problem for smaller, memory-bound cases. In fact, NBP has been so designed that a subset is all that is needed for such servers. This subset simply responds to "lookup" packets received over the network. Since the names table of such a server will contain only a single name, this NBP subset need not implement functions such as names table management, etc.


## Network Number Equivalence

The use of network number zero to indicate "unknown" introduces some complications for DDP clients. A DDP client may wish to compare two network numbers to determine if they are equivalent. For instance a request may be sent to a node on network 7 and a response received from one on network zero. Was the response received from the network to which the request was sent? For this reason, we must somewhat arbitrarily define when two network numbers match (i.e. are equivalent). The rule to use is "zero matches anything." Thus network A is equivalent to network B if A=B or A=0 or B=0. All DDP clients should use this definition of network equivalence.
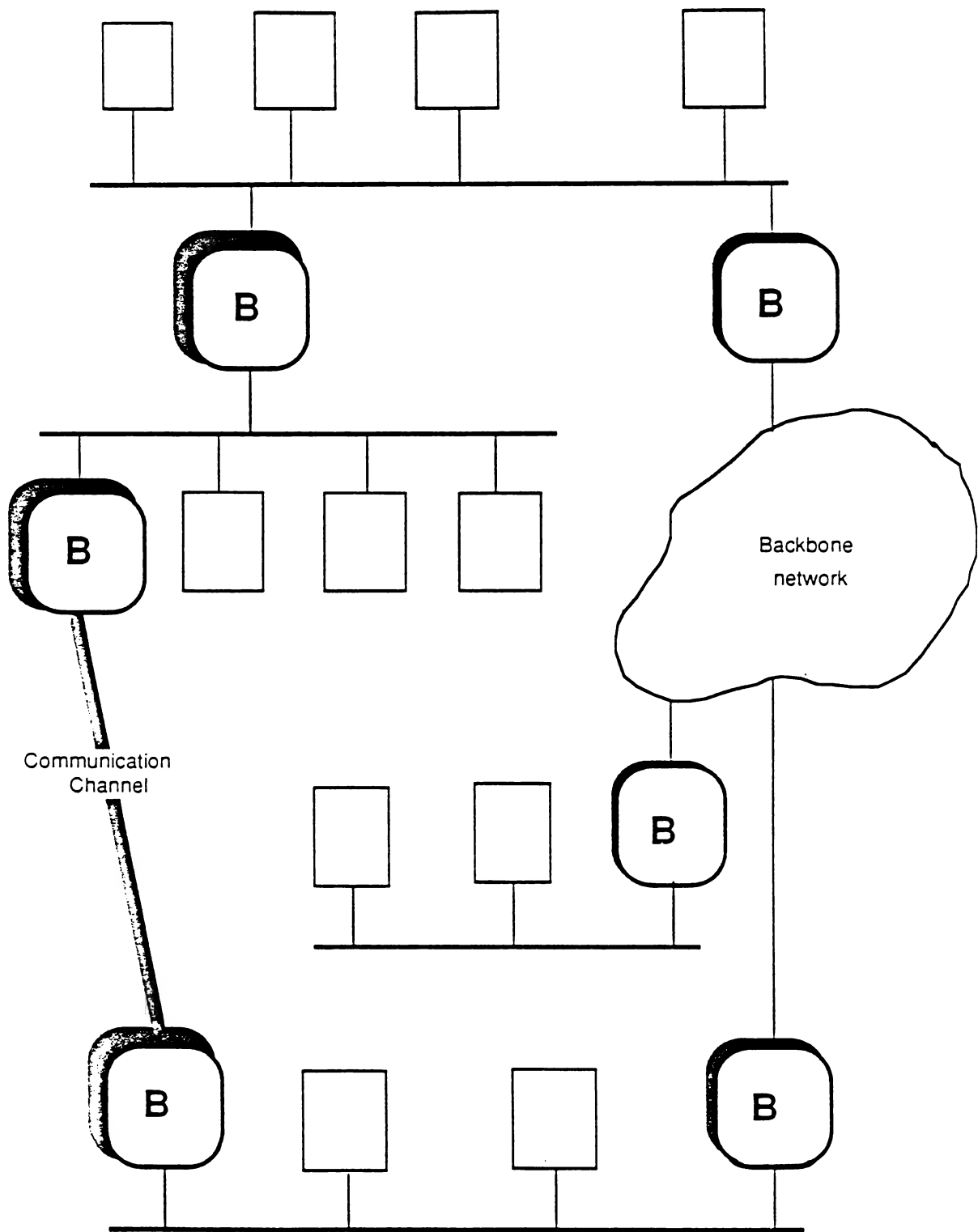
Figure IV-1: AppleTalk internet and
bridges/internet-routers (B)
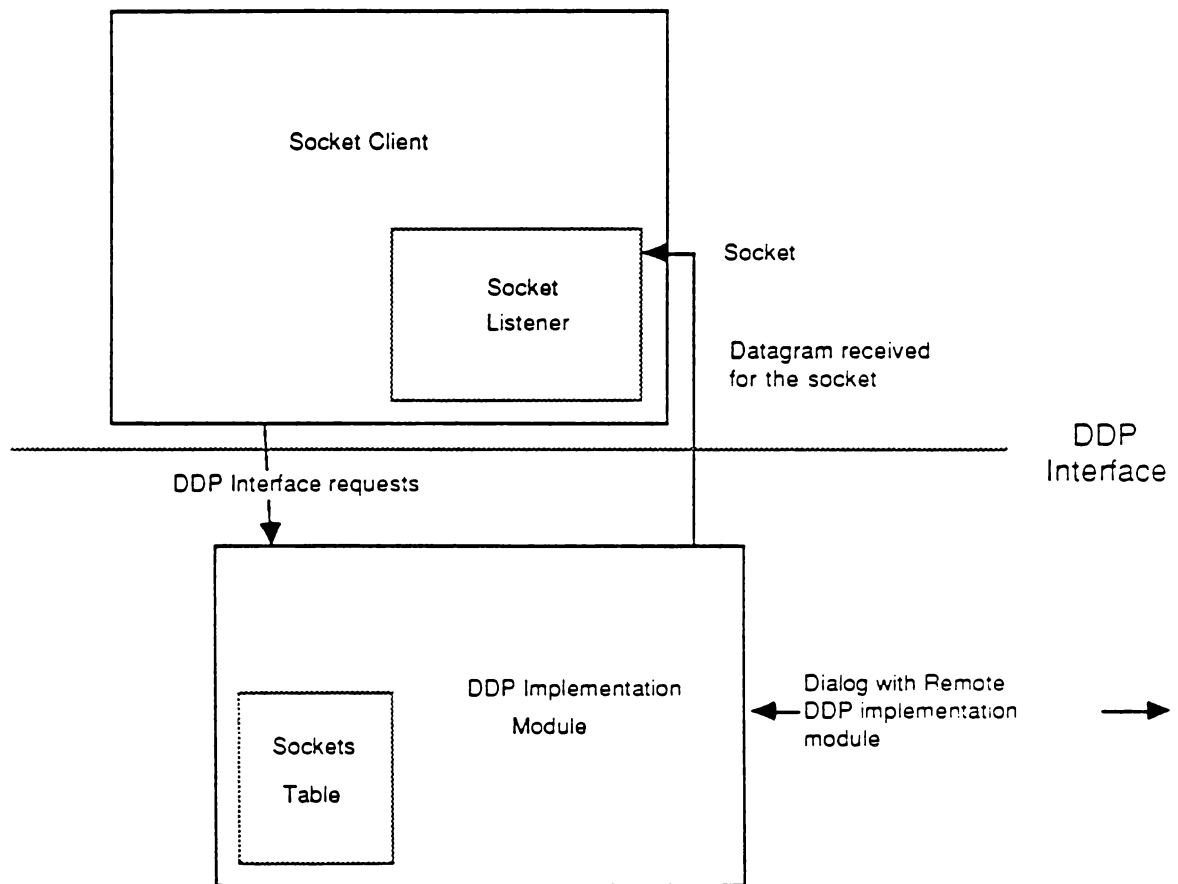
**Figure IV-2: Socket Terminology**

One Byte (8 bits)

LAP Header

LAP type field = 1

| 0 0 0 0 0 0 | DL (msb) |

Datagram length (lsb)

Destination Socket Number

DDP Header

Source Socket Number

DDP Protocol Type

Datagram Data
(0 to 586 bytes)
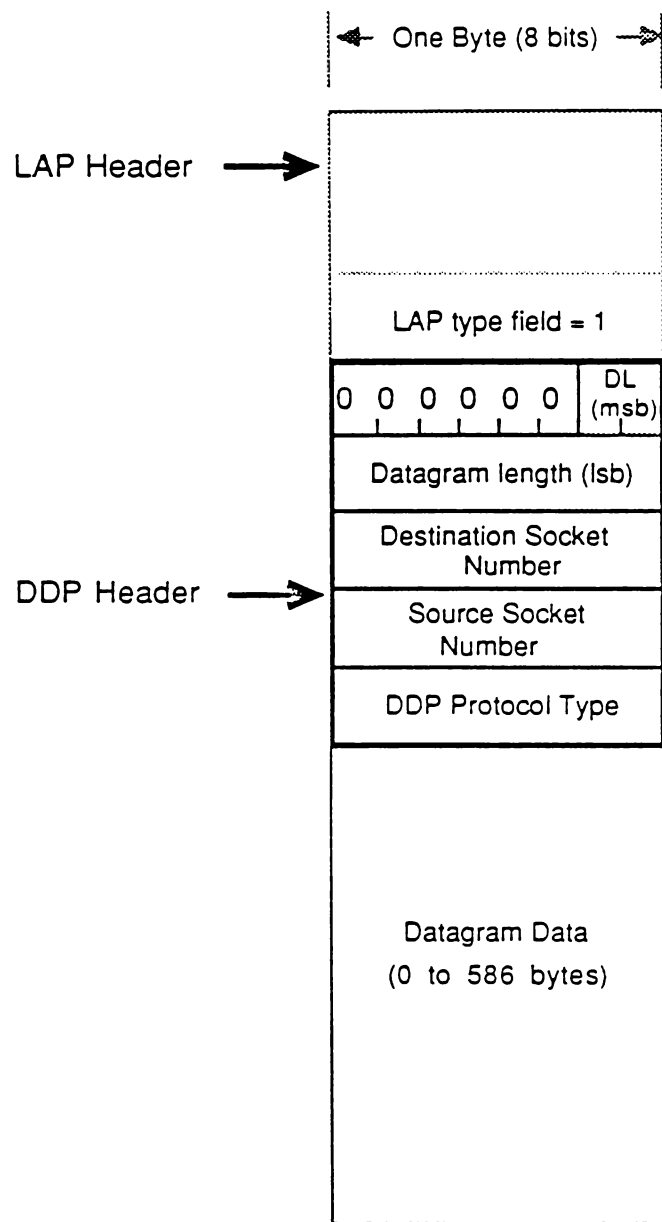
Figure IV-3: DDP Packet Format (Short Header)

```
                      ⟵ One Byte (8 bits) ⟶

                     ┌────────────────────────┐
                     │                        │
LAP Header  ⟶        │                        │
                     │                        │
                     ├────────────────────────┤
                     │    LAP type field = 2   │
                     ├──────┬──────────┬───────┤
                     │      │          │  DL   │
                     │ 0  0 │hop count │ (msb) │
                     │      │          │       │
                     ├──────┴──────────┴───────┤
                     │  Datagram length (lsb)  │
                     ├────────────────────────┤
                     │                        │
                     ─     DDP Checksum      ─
                     │                        │
                     ├────────────────────────┤
                     │                        │
                     │      Destination       │
                     ─   Network Number     ─
                     │                        │
DDP Header  ⟶        ├────────────────────────┤
                     │                        │
                     │        Source          │
                     ─    Network Number    ─
                     │                        │
                     ├────────────────────────┤
                     │      Destination       │
                     │       Node ID          │
                     ├────────────────────────┤
                     │        Source          │
                     │       Node ID          │
                     ├────────────────────────┤
                     │      Destination       │
                     │    Socket Number       │
                     ├────────────────────────┤
                     │        Source          │
                     │    Socket Number       │
                     ├────────────────────────┤
                     │   DDP Protocol Type     │
                     ├────────────────────────┤
                     │                        │
                     │                        │
                     │     Datagram data      │
                     │   (0 to 586 bytes)     │
                     │                        │
                     │                        │
                     └────────────────────────┘
```
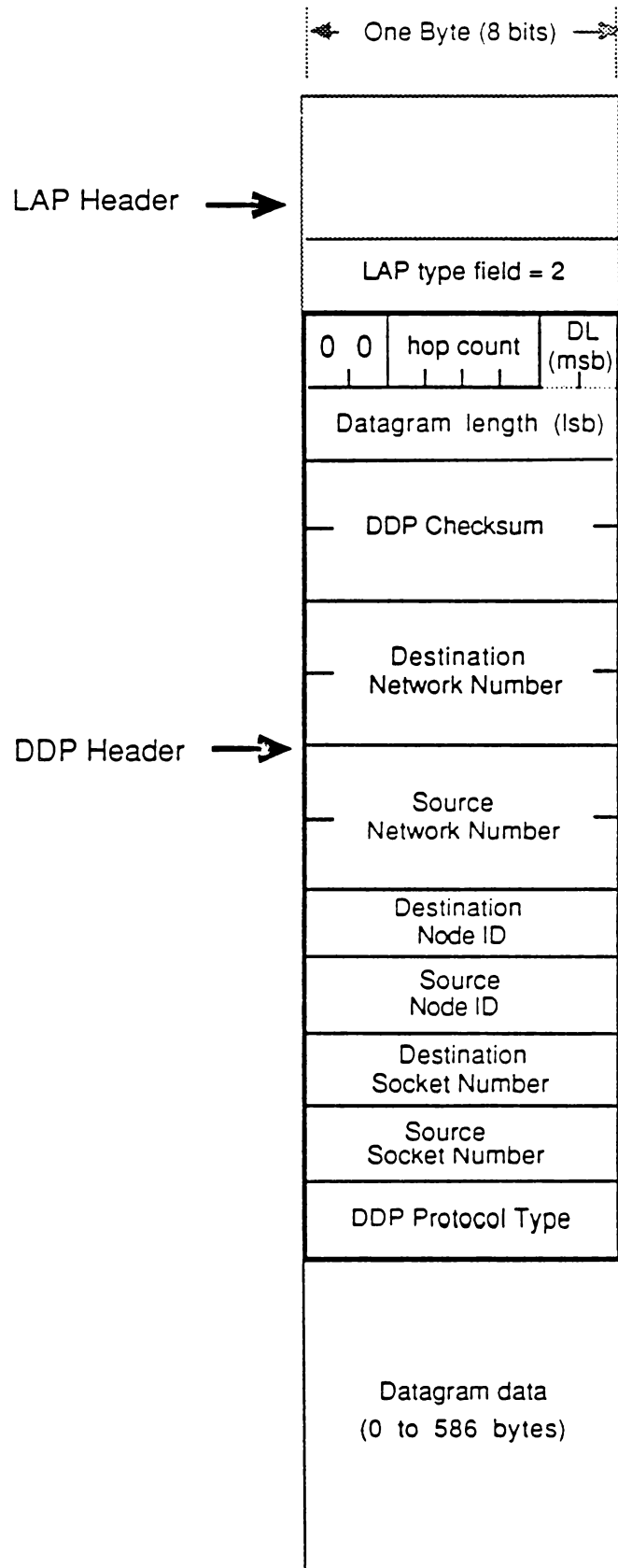
Figure IV-4:  DDP Packet Format (extended header)

# V. Routing Table Maintenance Protocol (RTMP)

## About Routing Table Maintenance Protocol

The Datagram Delivery Protocol introduced the concept of bridges or internet routers (IR) as the mechanism by which datagrams are forwarded/routed from any source socket to any destination socket on the internet. The Routing Table Maintenance Protocol (RTMP) is used by bridges to establish and maintain the *routing tables* that are central to this forwarding process.

## Bridges

Bridges are the key components in extending the datagram delivery mechanism to an internet setting. It is useful to distinguish the ways in which bridges can be used to build an internet; figure V-1 illustrates three principal situations.

### Local Bridges

Configuration A shows a bridge used to interconnect several AppleTalk networks that are in close proximity; such a bridge can be said to be a *local bridge*. Such local IRs are connected directly to each of the AppleTalk networks they serve to bridge. They are useful in allowing the construction, within the same building, of an AppleTalk internet with more than 32 nodes.

### Half Bridges

Configuration B illustrates the use of two bridges interconnected by a long distance communication link. Each bridge is directly connected to an AppleTalk network. The combination of the two bridges and the intervening link serves in effect as a bridging unit between the AppleTalk systems. Each bridge in this unit will be referred to as a half bridge. The intervening link can of course be made up of several devices (such as modems), and other networks (such as public data networks, etc.). The primary use of half bridges is to interconnect remote AppleTalk systems. An important characteristic of such bridges is that the throughput is generally lower than in the case of local bridges, due to the addition of the communication link. Furthermore, these communication links can often be less reliable than the local networks of the internet.

### Backbone Bridges

Although these bridges might be placed in one of the previous categories, they present an important set of properties that make it appropriate to single them out. These bridges are used to interconnect several AppleTalk networks through a backbone network. Each bridge could be a local bridge connected on one side to an AppleTalk network, and to the backbone network on the other. This is illustrated as configuration C. Another manner of connecting a backbone bridge to the backbone network might be through a long-distance communication link.

## Bridge Model

We model a bridge as a device with several hardware ports, referred to as *bridge ports* (see figure V-2). A bridge port can be connected in any of the three ways described above (local to an AppleTalk, half to a communications link, or backbone to a high-speed net), or even as a combination of all three. In our model a bridge can have any number of ports, which are numbered starting with the number 1.

Each bridge port has associated with it a *port descriptor*. This consists of four fields: (1) a flag as to whether the port is connected to an AppleTalk network or not, (2) the port number, (3) the port node ID, (the bridge's node ID corresponding to that port), and (4) the port network number, (the network number of the network to which the port is connected).

The values of these four fields are obvious for a port that is directly connected to a local AppleTalk network. For a port connected to one end of a communication link (half bridge case), the port node ID and port network number are meaningless. For a port connected to a backbone network, the port network number is meaningless, while the port node ID is the appropriate node ID of the bridge on the backbone network. In this latter case, provision must be made for this field to be of any size (possibly variable length) depending on the nature of the backbone network.

It is important to understand that the AppleTalk node ID of a local bridge is different for each of its ports. In other words, for each AppleTalk network to which it is directly connected, the bridge acquires a different node ID.

The bridge internals include a suitable data link process (ALAP or other) for each port, a (DDP) router, the routing table, and the routing table maintenance process (RTMP) implemented on a statically-assigned socket known as the RTMP socket (socket number = 1). The router accepts incoming datagrams from the LAPs and then reroutes them out through the appropriate port depending on their destination network number. This routing decision is made by the router by consulting the routing table. The routing table maintenance process receives RTMP packets, from other bridges, through the RTMP socket and uses these to maintain/update the routing table. (Bridges additionally consist of a Zone Information Protocol process and an NBP routing process; these are discussed in the ZIP chapter).

## Internet Topologies

RTMP allows internets to consist of AppleTalk networks interconnected via bridges in any arbitrary topology. The only strict limitation imposed on an AppleTalk internet is that, for each bridge, no two of its ports be on the same network. In addition, nodes on any network which is more than sixteen hops away (via the shortest path) from any other network will not be able to communicate with nodes on that network.

## Routing Tables

All bridges maintain complete routing tables that allow them to determine how to forward a datagram on the basis of its destination network number. RTMP allows bridges to periodically exchange their routing tables; a bridge receiving the routing table of another bridge compares it with its own table, updating its own to record the shortest path for each destination network. This exchange process allows the bridges to respond to changes in

the connectivity of the internet (such as when a bridge goes down or a new bridge is installed).

A routing table that has stabilized to all changes will consist of one entry for each reachable network in the internet. Each entry provides the number of the port through which packets for that network must be forwarded by the bridge, the node ID of the next IR/bridge, and a measure of the "distance" to the destination network. The entry in the routing table corresponds to the shortest path (known to that bridge) for the corresponding destination network. The distance corresponding to a network to which the bridge is directly connected is always zero.

The distance to a network is measured in terms of "hops", each hop representing one IR/bridge that the packet will encounter in its path from the current bridge to the destination network. This simple measure of distance is adequate for an RTMP that adapts to changes in the connectivity of the network.

(Other modified measures could reflect the speeds/capacities of the intervening links and thus try to find a minimum time path. Yet another enhancement might use current traffic conditions on a particular path to modify its contribution to distance. We have, for sake of simplicity, chosen to use the hop-count measure. The basic algorithm remains unchanged if more complex measures are used.)

Each table entry has associated with it an *entry state value*. This is a variable which takes on one of three values: good, suspect, bad. The significance of the entry state will become clear when the table maintenance mechanism is discussed in more detail.

Figure V-3 illustrates a typical routing table for a bridge with three ports in an internet consisting of seven networks. The corresponding port descriptors are shown in the figure as well.

## Routing Table Maintenance

Since bridges have no knowledge of the topology or connectivity of the internet, RTMP must provide the mechanism for constructing their routing tables, and maintaining them in the face of changes in the internet (bridges coming up or going down).

When a bridge is switched on, it initializes its table by examining the port descriptor of each of its ports. Any AppleTalk port with a non-zero port network number signals that the bridge is directly connected to that network. Thus an entry is created in the table for that network number, with a distance of zero and with that port's number in the appropriate part of the entry. This initial table is called the *routing seed* of the bridge.

Every bridge must periodically broadcast one or more *RTMP data packets* through each of its ports, addressed as datagrams to the RTMP socket. Thus, every bridge's RTMP will periodically receive RTMP data packets from every bridge on its directly connected networks, backbones or communication links. The RTMP data packets (see figure V-4 ) carry the port node ID and port network number of the bridge port through which the RTMP data packet was sent, as well as the <network number, distance> pairs (called *routing tuples*) of the entries in the sending bridge's routing table. Using these RTMP data packets, RTMP adds to or modifies its own routing table.

The basic idea is that if an RTMP data packet (received by the bridge) contains a routing tuple for a network not in the bridge's table, then an entry is added for that network number

with a distance one larger than the tuple's distance. In essence, the RTMP data packet indicates that a route exists to that network through the packet's sender.

Likewise, if an RTMP data packet indicates a shorter path to a particular network than the one in the bridge's routing table (i.e. if the tuple distance plus one is less than the table entry's distance), then the corresponding entry must be modified to indicate the RTMP data packet's sender as the next IR for that network with the new distance. In fact, even if the paths are of equal length, the entry is modified (thus information remains as up to date as possible).

Clearly, this process allows for the growth and adaptation of routing tables to the addition of bridges/routes.

"Aging" Routing Table Entries

However, if bridges die or are switched off, the corresponding changes will not be discovered through the foregoing process. To respond to such changes, the entries in the routing tables must be "aged" and, in the absence of confirmation via new RTMP data packets, be declared "suspect" and later "bad". Bad entries are eventually purged from the routing tables.

Each entry in the routing table (corresponding to a network to which the bridge is not directly connected) was obtained from an RTMP data packet received by the bridge from the next IR for that network. RTMP considers such an entry to be valid for a limited time only (called the *entry validity time*). Before starting the validity timer, the bridge goes through its routing table and changes the state of every "Good" entry to "Suspect". An entry must be revalidated from a new RTMP data packet before the timer expires.

Suppose, for instance, that the next IR, for a particular entry in bridge B's routing table, dies. That IR will not be sending RTMP data packets; the entry's validity timer will expire and bridge B will not have received confirmation of the entry. At that time, the entry's state is changed from "Suspect" to "Bad". Any other RTMP data packet received with path information to the network of the entry can be used to replace the entry with the new values from that packet. If no new route is discovered, however, the bad entry will be deleted when the validity timer goes off a second time.

A more detailed discussion of this process is provided in the section "RTMP Algorithms" below.

Values for the Validity and Send-RTMP Timers

These values have a significant effect on the dynamics of the propagation of routing table adaptation in response to changes in the internet's connectivity, and on network traffic. The exact values of these parameters have been determined through experimentation with actual internets. These values are 10 seconds for the Send-RTMP timer and 20 seconds for the Validity timer.


## RTMP Data Packet Format

The format of an RTMP data packet is shown in figure V-4. Clearly, the datagram need use only the short form of the DDP header. The DDP type field is set to 1 to indicate that it is an RTMP data packet. The DDP data part of the datagram consists of three parts: the sender's network number and node ID, and the routing tuples.

Sender's Network Number

The first two bytes of the RTMP data packet's DDP data is the port network number from the port descriptor (of the port through which the packet is sent by the bridge). This field allows the receiver of the packet to determine the number of the network through which the packet was received. This is thus the number of the network to which the corresponding port of the receiver is attached. RTMP Data packets sent out ports which are not AppleTalk networks should have this field set to zero.

Sender's Node ID

This field contains the port node ID from the port descriptor (of the port through which the packet is sent by the bridge). To allow for the case of ports connected to backbone networks other than AppleTalk networks, this field must be of variable size. The first byte of the field contains the length (in bits) of the sender node's ID. This is followed by the ID itself. If the length of the ID in bits is not an exact multiple of 8, it is prefixed with enough zeros to make a complete number of bytes. The bytes of this modified ID are then packed into the ID field of the packet, most significant byte first. It is from this field that the receiver of the packet determines the ID of the bridge sending the packet.

Routing Tuples

The last part of the RTMP data packet consists of the routing tuples from the sending bridge's routing table. These <network number, distance> pairs consist of two bytes for the network number and one byte for the distance.

For internets with a large number of networks, the entire routing table may not fit in a single datagram. In that case, the tuples are distributed over as many RTMP data packets as necessary.

## Assignment of Network Numbers

Network numbers are set into the port descriptors of the bridge ports, and are then dynamically propagated out through RTMP to the other nodes of each network.

Not all bridges on a particular network must have the network number set into their corresponding port descriptors. The precise requirement is that at least one bridge (called the *seed bridge*) on a network should have the network number built into its port descriptor. The other bridges could have a port network number value of zero; they will acquire the correct network number by receiving RTMP data packets sent out by the seed bridge.

An absolute requirement is that the bridges on a particular network should not have (in their port descriptors) conflicting port network numbers for that network. (The value zero does not cause a conflict).

If a bridge is not a seed bridge for a particular port, it should not send its routing table out that port, nor should it have an entry in its routing table for that port, until it acquires that port's network number. It must, however, operate correctly with regard to those ports whose network number it does know.

## RTMP Initialization and Maintenance Algorithms

### Initialization

A bridge upon being switched on performs the following algorithm:

```
FOR each port P connected to an AppleTalk
    IF the port network number <> 0 THEN
        create an entry for that network number:
            Entry's network no. = port network number
            Entry's distance = 0
            Entry's next IR = 0
            Entry's state = Good
            Entry's port = P;
```

This algorithm creates an entry for each directly-connected AppleTalk for which the bridge is a seed bridge.

### Maintenance

The bridge is assumed to have two timers running continuously. These are the *validity timer* and the *send-RTMP timer*. The events to which the bridge's RTMP process responds are: RTMP data packet is received, the validity timer expires, the send-RTMP timer expires. The corresponding algorithms are given below:

(1) RTMP data packet is received through port P:

```
IF P is connected to an AppleTalk AND P's network number = 0 THEN
    BEGIN
        P's network number := packet's sender network number;
        IF there is an entry for this network number THEN delete it;
        Create a new entry for this network number:
            Entry's network no. = packet's sender network no.
            Entry's distance = 0
            Entry's next IR = 0
            Entry's state = Good
            Entry's port = P;
    END;

FOR each routing tuple in the data packet DO
    IF there is an entry in the table corresponding to the tuple's network number
    THEN Update-the-Entry
    ELSE Create-New-Entry;
```

General-purpose routines: Update-the-Entry, Create-New-Entry, and Replace-Entry are as follows:

## Update-the-Entry

```
        IF (Entry's State = Bad) AND (tuple distance < 15) THEN
                Replace-Entry
        ELSE
                IF Entry's distance >= (tuple distance+1) AND (tuple distance < 15) THEN
                        Replace-Entry
                ELSE
                        IF Entry's next IR = data packet's sender node ID AND Entry's port = P THEN
                        { If entry says we're forwarding to the IR who sent us this
                        packet, the net's now further away than we thought }
                                BEGIN
                                        Entry's distance := tuple distance + 1;
                                        IF entry's distance < 16 THEN Entry's state := Good
                                        ELSE Delete the entry
                                END;
```

## Create-New-Entry

```
        Entry's network number := tuple's network number;
        Replace-Entry;
```

## Replace-Entry

```
        Entry's distance := tuple's distance + 1;
        Entry's next IR := packet sender's ID;
        Entry's port number := P;
        Entry's state := Good;
```

## (2) Validity Timer Expires:

```
        FOR each entry in the routing table DO
          CASE Entry's state OF
                Good:   IF entry's distance <> 0 THEN Entry's state := Suspect;
                Suspect: Entry's state := Bad;
                Bad:    Delete the entry
          END;
```

(3) Send-RTMP Packet Timer Expires:

```
IF  routing table is not empty THEN
   BEGIN
          Copy the network number and distance pairs of each Good or Suspect
          entry of the routing table whose distance < 15 into the routing tuple fields
          of the RTMP data packet;

          FOR each bridge port DO
            IF the port is connected to an AppleTalk AND its network no. is non-zero THEN
              BEGIN
                   Packet's sender network number := port network number;
                   Packet's sender node ID := port node ID;
                   Call DDP to broadcast the packet to the RTMP socket;
              END
            ELSE
            IF the port is not connected to an AppleTalk THEN
              BEGIN
                   Packet's sender network number := 0;
                   Packet's sender node ID := port node ID;
                   Call DDP to broadcast the packet through the port's LAP
                           to the RTMP socket;
              END;
   END;
```

## Supplements to the Algorithm for Half-Bridges

The above algorithm can be used for half-bridges, where the communications link between the half-bridges is treated as a network. In this case, the communications link will count as an additional hop. Due to the fact that this link is generally of limited bandwidth (e.g. 9600 baud), sending of large routing tables between half-bridges may waste a good portion of that bandwidth, leaving little for the actual routing of packets. As an alternative, the two half-bridges plus the communications link, taken as a unit, can be considered logically to be one local bridge. In this case there is, logically, only one routing table (and Zone Information table), which is distributed between the two half-bridges. The communications link is used, not as a network, but as a medium to maintain this distributed information.

An example of this idea follows. Each half-bridge maintains a complete copy of the routing table. Whenever a half-bridge receives information (through one of its ports not connected to a communications link) that results in a change to that table, it sends only the change over the communications channel. In this way, under stable internet conditions, none of the channel will be wasted with routing information. Steps, of course, must be taken to insure the consistency of this distributed data base, for example an acknowledgement which includes some sort of checksum of the information and the ability to re-initialize one or both sides and start from scratch.

Half-bridges must accept and process, as a minimum, RTMP Data packets as will be sent when the communications link is considered to be a network (they must also accept and process normal ZIP packets). Many half-bridges, however, will also wish to implement schemes as described above. Packets sent to implement this type of scheme should be differentiated from normal AppleTalk packets through a *non-DDP LAP type* (neither 1 nor 2). Bridges which do not understand these packets should ignore them.

## Additional RTMP Services

RTMP provides a service which enables non-bridge nodes to request the number of the network they are on and the address of a bridge on that network. This service would normally be used by a node when it was first turned on. The non-bridge nodes makes the request by broadcasting, through any socket, a DDP packet of protocol type 5, addressed to the RTMP listening socket (socket 1). This packet is known as an *RTMP request*.. The response, sent by any bridge receiving the request, takes the form of a normal RTMP data packet, except that it is sent directly to the requesting node and it contains no routing table information. Thus the non-bridge node can receive the response through the exact same mechanism it uses to monitor RTMP data packets (see the following section). The specific formats of the request and the response are provided in figure V-4.

RTMP and the Zone Information Protocol (ZIP) are integrally related. ZIP uses both the routing table information and many of RTMP's significant events in its operation. ZIP also requires an additional RTMP service for the purpose of changing the zone name of a network. This service is described in detail in the ZIP chapter. Basically, ZIP must be able to request RTMP to remove an entry for a directly connected network from its routing table, and to replace that entry at a future time. The ZIP chapter details how these requests work.

## RTMP and Non-Bridge Nodes

Non-bridge nodes do not need to maintain routing tables. As noted in the DDP chapter, these nodes only need to know the number of the network to which they are connected (THIS-NET) and the node ID of any bridge on that network (A-BRIDGE).

When such a node first comes up, the values of both of these variables are zero ("unknown"). These nodes can obtain the correct values dynamically by listening for RTMP data packets being sent out by the bridges on the network (if they choose to do this, they should be sure to wait long enough to get such a packet). Alternatively, they can broadcast out an RTMP request to obtain that information. In either case, to receive this information, these nodes implement a very trivial RTMP process, known as the *RTMP stub*. This process "sits" on the RTMP socket in that node, and upon receiving an RTMP data packet, copies the packet's sender network number and sender node ID fields into THIS-NET and A-BRIDGE respectively. This is done every time an RTMP data packet is received. While THIS-NET will stabilize to a constant value, A-BRIDGE may change continually (if there is more than one bridge on the network).

Additionally, such nodes must maintain a background timer for the purpose of aging this information. This is to handle the case where all the bridges connected to a given network go down and the network becomes isolated. In this case, A-BRIDGE should be reset to zero. THIS-NET, however, should not be reset. The value of this timer should be on the order of 90 seconds. Each time an RTMP data packet is received, the timer is reset. If the timer ever expires, A-BRIDGE should be aged.

Note that it is important for the RTMP stub to differentiate between RTMP data packets, broadcast by bridges, and RTMP requests, broadcast by non-bridge nodes. The RTMP stub should ignore RTMP requests. This can be done because RTMP requests have a DDP protocol type of 5, whereas RTMP data packets have a DDP protocol type of 1.

## RTMP Routing Algorithm

The routing algorithm used by bridges to forward internet datagrams is given in figure V-5. This discussion applies only to the forwarding of packets received by the bridge through one of its ports, and does not hold for packets generated within the bridge. The algorithm assumes that when a packet is received through one of the bridge ports, it is tagged with the number of the port and placed in a queue. The router takes packets off this queue, and then executes the algorithm.

**Configuration A**

**Configuration B**

Communication Link

Local Bridge

Half Bridge

Half Bridge

**Configuration C**

Bridge

Backbone Network

Bridge

Bridge

**Configuration D**

Backbone Network
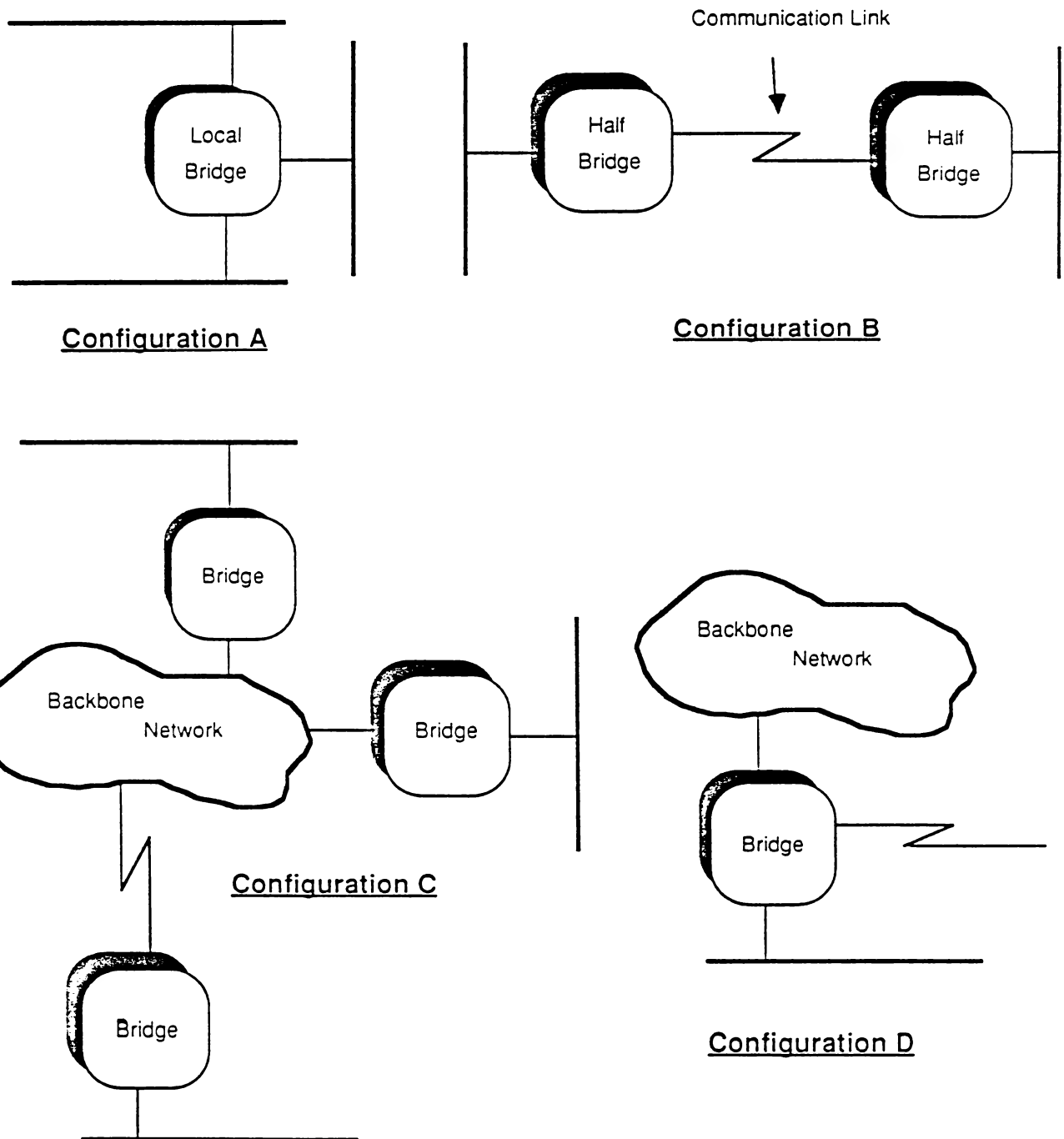
Bridge

Figure V-1. Bridge configurations
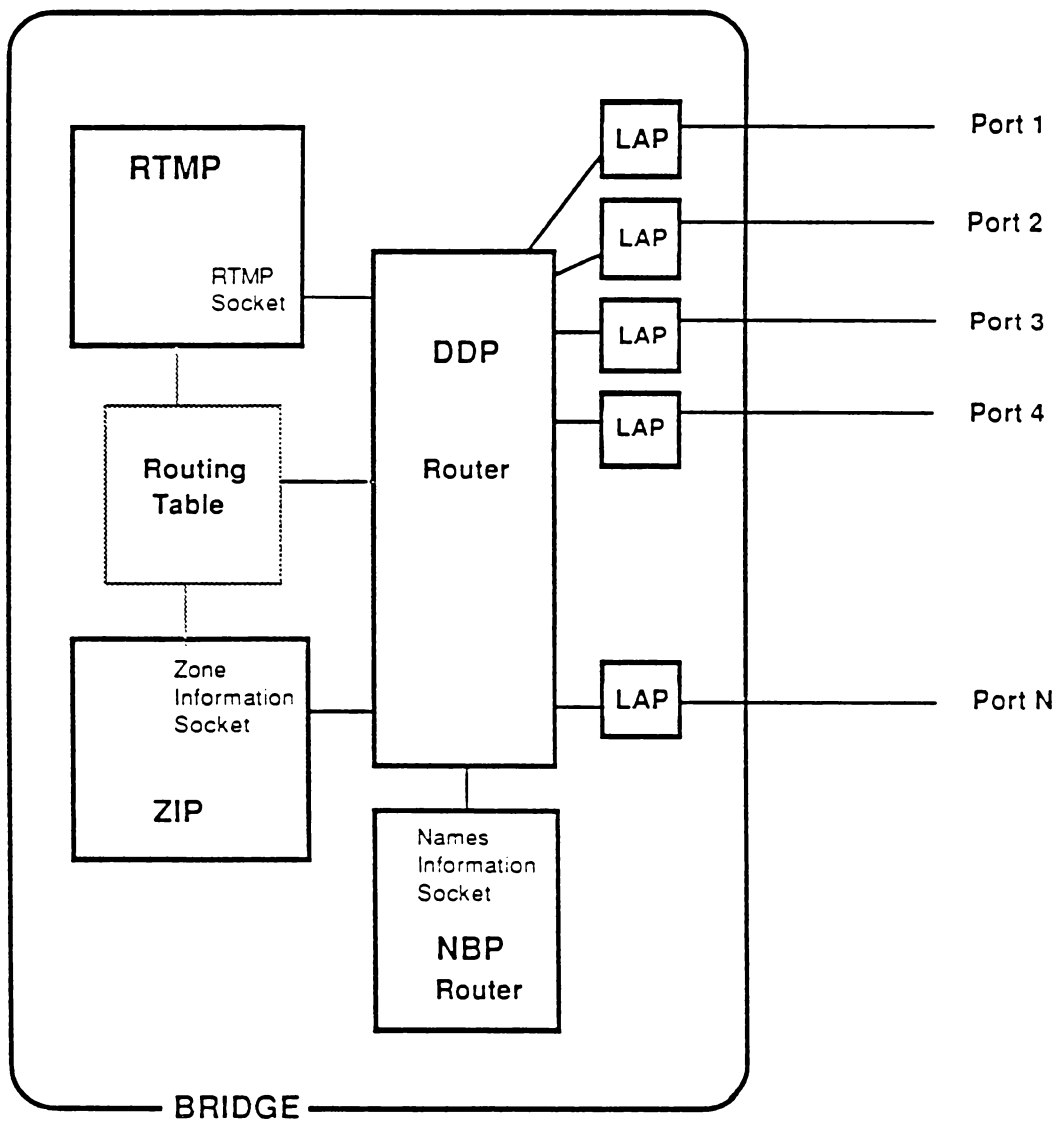
Figure V-2. Bridge model
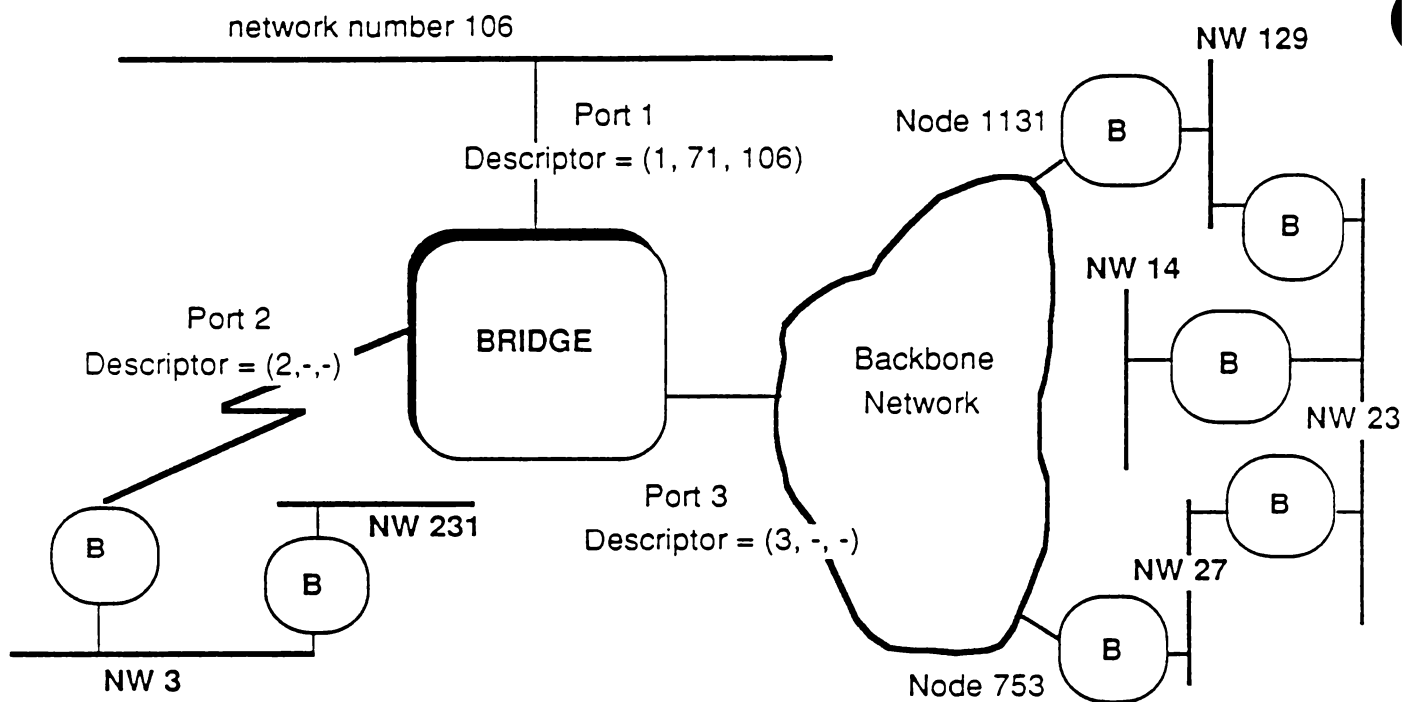
| Network Number | Distance | Port | Next Bridge/IR | Entry State |
|---|---|---|---|---|
| 3 | 1 | 2 | 0 | Good |
| 14 | 3 | 3 | 1131 | Good |
| 23 | 2 | 3 | 753 | Good |
| 27 | 1 | 3 | 753 | Good |
| 106 | 0 | 1 | 0 | Good |
| 129 | 1 | 3 | 1131 | Good |
| 231 | 2 | 2 | 0 | Good |

Figure V-3. Example of a Routing Table

RTMP Request                    RTMP Response                    RTMP Data Packet

| LAP Dest = SFF |
| LAP Source (n) |
| LAP Type = 1 |
| —  Length  — |
| Dest Socket = 1 |
| Source Socket |
| DDP Type = 5 |
| Command = 1 |

| LAP Dest = n |
| LAP Source |
| LAP Type = 1 |
| —  Length  — |
| Dest Socket = 1 |
| Source Socket = 1 |
| DDP Type = 1 |
| Sender's Network number |
| ID Length |
| Sender's ID |

Node ID of the bridge sending out the RTMP packet

Routing Tuples from the sending bridge's routing table

| LAP Dest = n |
| LAP Source |
| LAP Type = 1 |
| —  Length  — |
| Dest Socket = 1 |
| Source Socket = 1 |
| DDP Type = 1 |
| Sender's Network number |
| ID Length |
| Sender's ID |
| —  Network Number  — |
| Distance |
| |
| —  Network Number  — |
| Distance |

Figure V-4: RTMP Packet Formats

Is packet's LAP type = 1?
(Short DDP header)

——no——

——yes——

Is DDP dest net = reception port's net
and
DDP dest node ID = broadcast or reception port's node ID

——yes——

no

Search for Routing Table entry
corresponding to packet's DDP dest.

——found——

Entry's distance = 0?

no

yes

DDP dest. node ID = table
entry's port node ID?

——yes——

not found

——no——  Packet's hop count < 15?

no

DDP dest. node ID = broadcast?

yes

no

yes

Increment packet's hop count;
Send to table entry's next IR
through table entry's port

Deliver a copy to dest.
socket in this bridge

Send to destination node through
routing table entry's port

Discard the packet

{Packet addressed to socket in bridge}
Deliver to appropriate socket

Figure V-5 :  Datagram routing algorithm for a bridge

# VI. Name Binding Protocol (NBP)

## About Name Binding Protocol

AppleTalk protocols rely on numerical identifiers, such as node IDs, socket numbers and network numbers, to provide the basic addressing capability essential for communication over a packet-switched network. However, human users may not find numbers the most convenient form of identification. Numbers are hard for them to memorize, and are easily confused and misused. Names are more easily used by the human user. Thus, if they are allowed to refer to objects by their names, the name must be converted into a network address for use by the other protocols. The Name-Binding Protocol (NBP) performs this conversion.

AppleTalk uses dynamic node ID assignment, which does not allow addresses to be configured into software for accessing network resources. The name-binding approach provides the preferable solution towards this end (see the section "Use of Name-Binding Protocol in Conjuntion with Sockets" in the Datagram Delivery Protocol chapter).

## Network-Visible Entities

We start by defining the concept of a *network-visible entity* (NVE). In general terms, this is any entity that can be accessed over the network. More specifically, the socket clients on an internet are its network-visible entities.

The nodes of the network are not network-visible entities; rather, any services in the nodes available for access over the network are network-visible entities. Consider for instance, a print server on a network. The server is not the network-visible entity. The print service will typically be a socket client on what might be called the server's request listening socket. The latter is the server's network-visible entity.

The same distinction applies to the human users of the network. They are not network-visible themselves. But a user may have an electronic mailbox on a mail server. This mailbox is network-visible and will have a network address. This distinction is quite valid, since the network does not provide any protocols for conversing with the individual user, but rather to certain applications/services through which the user may be accessed.

## Entity Names

Network-visible entities may assign names to themselves, though not all NVEs have names. Such a name will be called simply an *entity name*. Entity names are character strings. A particular entity could in fact possess several names (*aliases* ).

In addition to a name, an entity could also have certain attributes. For instance, a print server's request receiver might have associated with it a list of attributes of the printer, such as its type (daisy wheel, dot matrix, laser), the type and size of paper, etc. So the mapping of the entity name into its socket address might be complemented with some form of entity attributes "look-up" service.

In defining the permissible structure of an entity name, we have decided to code the attributes into the name as a separate field, known as the entity type.

In addition to attributes, it might prove useful to have some logically-defined location information about the entity. For instance, a given file server might belong to a particular department or building. The users of the network should be able to select a file server on the basis of its being in the vicinity or in their department, etc. Thus, the concept of a zone has been defined, and a zone field has been included in the entity name. A precise definition of the concept of a zone is provided below.

Thus, an entity name is a character string consisting of three fields: object, type, and zone concatenated together in this order with colon and @-sign separators. For example, Sidhu:Mailbox@Bandley3. Each field is an alphanumeric string of at most 32 characters. Note that by definition, all fields are case insensitive.

Certain meta-characters can be used in place of explicit strings. For the object and type fields, an '=' can be substituted, signifying 'all possible values' (wildcard). For the zone field, a '*' may be substituted, meaning the default value (the zone in which this node resides). If a network name does not contain any meta-characters, it is said to be fully specified. As an example, =:Mailbox@* refers to all mailboxes in the same zone as the information requester. Likewise, =:=@* would mean all named entities of all types in the requester's zone. Again, Sidhu:=@* refers to all entities named Sidhu in the requester's zone regardless of their type.


## Name Binding

Before a named entity can be accessed over an AppleTalk network or internet, the address of that entity must be obtained. This is done by a process known as name-binding.

Name binding may be visualized as a mapping of an entity name into its internet address, or equivalently, as a lookup of the address in a large data base, etc. (We will refer to the entity's internet address; this includes the case of a single network, where the network number field will always be equal to zero -- "unknown").

Name binding can be done at various times. One strategy is to configure the address of the named entity into the system trying to access that entity. This so-called static binding is not appropriate for systems such as AppleTalk where the node ID can change every time a node comes up.

It is useful in this context to remember that although entities can move on a network, their names seldom change. For this reason, it is best to configure names into systems, and then use services such as NBP to bind dynamically. This may be done when the user/accessor's node is first brought up (known as early binding) or just before the access to the named entity is performed (known as late binding). Early binding runs the risk of using incorrect (out-of-date) information when the resource is actually accessed, possibly long after the user's node was brought up. However, since the binding process might be a slow one, late binding might impinge upon the response obtained by the user when accessing the named entity. Late binding is the method most appropriate when the entity is expected to "move" on the internet.

### Names Directory and Names Tables

Each node maintains a *names table* (NT) containing entity name-to-entity internet address mappings (known as *NBP tuples*) of all entities in that node. Name binding is done on AppleTalk by using NBP to look up the entity's address in the *names directory* (ND). The

ND is a distributed data base of entity name-to-entity address mappings; it is the union of the individual names tables in the nodes of the internet.. The data base does not require different portions to be replicated. It can be distributed among all nodes containing named NVEs. NBP places no restriction on the use of name servers, nor are such servers necessary.

<u>Aliases and Enumerators</u>

NBP allows an NVE to have more than one name. Each of these aliases must be included in the NT as an independent entity.

To simplify and speed up the ability to distinguish between multiple names associated with a given socket, an enumerator value is associated with each NT entry. This is a one-byte integer, totally invisible to the clients of NBP. Each NBP implementation can develop its own scheme for generating enumerator values to be included in its NT, subject to the condition that no two entries corresponding to the same socket have the same enumerator value.

<u>Names Information Socket</u>

Each node implements an NBP process on a statically-assigned socket (socket number = 2) known as the names information socket (NIS). This process is responsible for maintaining the node's names table, and for accepting and servicing name lookup requests from within the node and from the network (through the NIS).

## Name-Binding Services

The name-binding service provides four basic services, described below.

<u>Name Registration</u>

Any entity can enter its name and socket number into the ND (actually into its node's NT) to make itself "visible by name". This is done by using the name registration call to the node's NBP process.

This process must first verify that the name is not already in use. This is done by performing a name lookup in the node's zone. If the name is already in use, the registration attempt is aborted. Otherwise, the name and the corresponding socket number are inserted into the node's NT. This enters the corresponding name-to-address mapping in the ND.

When a node comes up, its names table is empty. Each network-visible entity must re-register its name(s) when restarted.

<u>Name Deletion</u>

A named entity should delete its name-to-address mapping from the ND when it wishes to make itself "invisible". Reasons for doing so range from the obvious one of the entity wishing to terminate operation to sophisticated entity-specific control requirements.
The entity issues a name deletion call to the node's NBP process. The latter simply deletes the corresponding name-to-address mapping from the node's NT.

## Name Lookup

Before accessing a named entity, the user (or a surrogate application) must perform a binding of the entity's name to its internet address. This is done by issuing a name lookup call to the user node's NBP process. The latter then uses NBP to perform a search through the ND for the named entity. If it is found, then the corresponding address is returned to the caller. Otherwise an "entity not found" error condition is returned.

It is possible to find more than one entity matching the name specified in the call. This is especially true when the name includes the '=' wildcard. The interface to the user must have provisions for handling this case.

Another feature of NBP is that it does not allow abbreviated names; for instance, it does not permit reference to Sidhu:Mailbox. The complete reference Sidhu:Mailbox@Bandley3 or Sidhu:Mailbox@* is required by NBP. Provisions may be made in the user interface to permit abbreviations; the interface must then "flesh out" the name before passing it on to NBP.

## Name Confirmation

Name lookup performs a zone-wide ND search. More specific confirmation is needed in certain situations. For instance, if early binding was performed, the binding must be confirmed when the named entity is actually accessed. For this purpose, NBP has a name confirmation call in which the caller provides the full name and address of the entity. This call in effect performs a name search in the entity's node to confirm that the mapping is still valid.

Although a new name lookup can lead to the same result, the confirmation is more efficient in terms of total network traffic generated. It is the recommended and preferable call to use when confirming mappings obtained through early binding.

## NBP on a Single Network

Name searching/look-up is quite simple on a system consisting of a single AppleTalk network.

When a user issues a name registration or lookup request to the NBP process in its node this process first examines its own names table to determine if the name is available there. If so, in the case of a registration attempt, the call is aborted with a "name already taken" result. In the case of a name lookup, the information in the names table is a partial response (there may be entities in other nodes that match the specified name).

NBP then prepares an NBP lookup packet (LkUp packet ) and then calls DDP to broadcast it over the network for delivery to the Names Information Socket. Only nodes that have an NBP process will have this socket activated. In these nodes, the LkUp packet is delivered to the NBP process which searches its names table for a potential match. If no match is found, the packet is ignored. If a match is found, a LkUp-Reply packet is returned to the address from which the LkUp packet was received. This LkUp-Reply packet contains the matching name-to-address mapping (tuples) found in the replying node's names table.

The receipt of one or more replies allows the requesting NBP process to compile a list of name-to-address mappings for the original user. If the lookup was performed in response to a name registration call, then the call must be aborted since the name is already taken.

Since broadcasts are not very reliable, the requesting NBP process sends the LkUp packet several times before returning the compiled mappings, if any, to the requesting user. If no replies are received, then it is concluded that there is currently no entity using the specified name. For a name registration call, the requested name-to-address mapping is entered into the node's names table. For a name lookup call, the user is informed of a "no such entity" result.

Sending the LkUp packet several times implies that the same name-to-address mapping could be received by the requesting node several times in LkUp-Reply packets. These duplicates must be filtered out of the list of mappings. The obvious way is to compare the name strings and the address fields with each entry in the compiled list; this method, however, is inefficient. Comparison of the four-byte address fields is insufficient because of the possibility of aliases. Using the enumerator value together with the address resolves this problem, and accelerates the filtering of duplicates.

Name confirmation is similar in nature, except that the caller provides the name-to-address mapping to be confirmed. The LkUp packet is not broadcast, but is sent directly to the NIS at the specified address. This can be repeated several times to protect against lost packets or the target node being temporarily busy.

Note that on a single AppleTalk, there is only one zone. This zone should be considered to be an unnamed one (zone names originate in bridges). Any request made by a client to perform a lookup with a zone name other than '*' should be rejected with an error.

## NBP on an Internet

The use of broadcast packets to perform name searching is inconvenient in internets. DDP does not allow a destination address corresponding to a broadcast to all nodes in the internet. DDP can broadcast datagrams to all nodes of any single specified network in the internet (these are said to be directed broadcasts). If NBP sent a directed broadcast to every network in the internet, the traffic generated would be considerable. For this reason, the concept of zones has been introduced.

### Zones

A zone is an arbitrary subset of all the AppleTalk networks in an internet. A particular network can belong to one and only one zone. The networks in a particular zone need not be in any way contiguous or "neighbors". The union of all zones is exactly the internet.

The concept of zones is provided to assist the establishment of departmental or other user-understandable groupings of the entities of the internet. Zones are intelligible only to the NBP (and to the related Zone Information Protocol discussed in Chapter VIII). A zone is identified by a string of at most 32 characters.

### Name Lookup on an Internet

Bridges participate in the name lookup protocol of an internet. The NBP process in the requesting node prepares an NBP "broadcast request" packet (called a BrRq packet) and sends it to the NIS of A-BRIDGE. The NBP process in the bridge, in cooperation with the NBP processes in the other bridges of the internet, arranges to convert the BrRq packet into one LkUp packet for each network in the target zone of the lookup request (the exact details of this algorithm are specified in the ZIP chapter). Each of these LkUp packets is then sent

to the NIS socket as a directed broadcast on the corresponding network. The replies are returned to the original requester as before.

The important point is that non-bridge nodes do not need to know anything about zones. The process of generating a zone-wide broadcast is done by the collection of bridges. The latter must of course jointly have a complete mapping of zone names into the list of corresponding networks. The establishment and maintenance of these mappings is the purpose of the Zone Information Protocol discussed in Chapter VIII.

Note that on an internet, nodes performing lookups in their own zone will receive LkUp packets from themselves (via a bridge). The node's NBP process must not respond to these!

Note that the name registration process of NBP tries to ensure that the same name is not used by more than one entity in a given zone. However, if another network is "moved" into the zone by changing its zone name, then it is possible to end up with two or more entities (in the new zone) with the same name.

## NBP Interface

Four calls provide to the user all the functionality of name-binding; they are described below.

Note that all calls to NBP take as a parameter an entity name. It is only the lookup call, however, where the zone name is meaningful. In all other calls, it is required, for consistency, that the zone name be set to '*'.

Registering a Name

This call is used by an NBP client to register an entity name and its associated socket address. Meta-characters are not allowed in the object and type fields; the zone name field should be equal to '*'.

• Call Parameters:

> entity name
> socket number

• Returned Parameters:

> result code: success
>              failure (name conflict, invalid name or socket)

Although this is not a required feature of NBP it is advisable for the implementation of this call to verify that the socket is in fact open.

Removing a Name

This call removes an entity name from the node's names table. Meta-characters are not allowed in the object and type fields; the zone name field should be equal to '*'.

• Call Parameters:

entity name

• Returned Parameters:

>   result code: success
>   >   failure (name not found)

## Name Lookup

This call performs the mapping between entity name and address. Meta-characters are allowed in the name to make the search as general as needed. It is possible for more than one address to match the call's entity name. For each match, this call returns the name and its internet address. The names contain fully specified object and type fields; the zone name, however, will always be '*', regardless of zone specified.

• Call Parameters:

>   entity name
>   maxMatches

• Returned Parameters:

>   result code: success
>   >   failure (name not found)
>   list of entity names and their corresponding internet addresses

The parameter maxMatches is a positive integer that specifies the maximum number of matching name-to-address mappings needed. This is useful if wildcards are used by the caller in the entity name parameter.

## Confirming a Name

This call confirms a caller-supplied mapping between entity name and address. Meta-characters are not allowed in the object and type fields; the zone name field should be equal to '*'.

• Call Parameters:

>   entity name,
>   socket address

• Returned Parameters:

>   result code: success (mapping still valid)
>   >   wrong-socket (net and node number valid, socket number invalid)
>   >   failure (mapping invalid)
>   new socket (only if wrong-socket error code)

## NBP Packet Formats

NBP packets are identified by a DDP protocol type field of 2. There are three different types of NBP packets: BrRq, LkUp and LkUp-Reply. The format is as shown in Figure

VI-1. It consists of an NBP header followed by the name-address pairs (known as NBP tuples); the various fields are described below.

## Control

The high-order four bits of the first byte of the header are used to indicate the type of NBP packet. The values are 1 for BrRq, 2 for LkUp and 3 for LkUp-Reply.

## Tuple Count

The low-order four bits of the first byte contain a count of the number of NBP tuples in that packet. BrRq and LkUp packets carry exactly one tuple (the name being looked up or confirmed). The tuple count field for these packets is always equal to 1.

## NBP ID

In order to allow for multiple pending lookup requests from a given node, an 8-bit ID is generated by the NBP process issuing the BrRq or LkUp packets. The LkUp-Reply packets must contain the same NBP ID as the LkUp or BrRq packet to which they correspond.

## NBP Tuple

The format of the NBP tuples, the name-address pairs, is shown in figure VI-2. The tuple consists of an entity's name, its 4-byte internet address, and a one-byte enumerator field. The address field appears first in the tuple. The fifth byte in a tuple is the enumerator field, followed by the entity name. This consists of three string fields: one each for the object, type and zone names. Each of these strings consists of a leading 1-byte string length followed by up to 32 string bytes. The string length represents the number of bytes/characters in the string. The three strings are concatenated without any intervening padding.

The enumerator field is included to handle the situation where more than one name has been registered on the same socket. It should be noted that NBP specifically permits this use of aliases (or alternately, the use of a single socket by more than one NVE). In this case, each alias is given a unique enumerator value, kept in the NT along with the name-address mapping. The enumerator field is not significant in a LkUp or BrRq packet, and is ignored by the recipient of these packets.

In BrRq and LkUp packets (which carry only a single tuple) the address field contains the internet address of the requester; this allows the responder to properly address the LkUp-Reply datagram. In a LkUp-Reply packet, the correct enumerator value must be included in each tuple. This value is used for duplicate filtering.

Since normal nodes (and hence their NBP processes) are usually unaware of zone names, specifically their own zone name, tuples in LkUp-Reply packets do not specify a zone name. The zone names in these tuples are always equal to '*', regardless of the zone in which the lookup is performed. The requestor will know the zone name of these responses, since it must be the zone he asked for in the lookup request.

Note that in a LkUp or BrRq request, a null zone name (length byte equal zero) is equivalent to '*'.
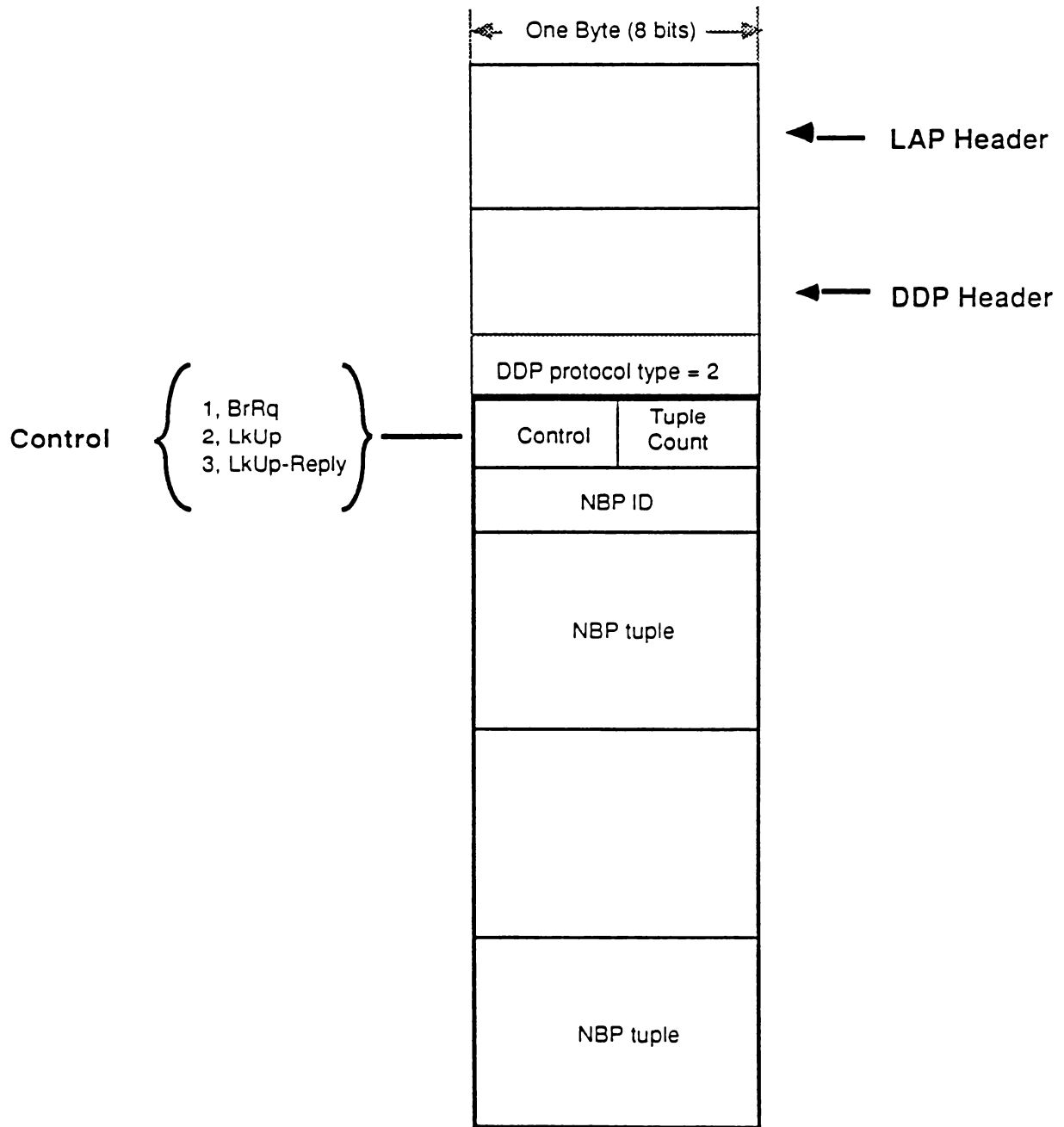
Figure VI-1: NBP Packet Format

```
|<--  One Byte (8 bits)  -->|

┌───────────────────────────┐  ⎫
│                           │  │
│        Network            │  │
│        Number             │  │
│                           │  │
├───────────────────────────┤  │
│         Node ID           │  ⎬  Entity Address
├───────────────────────────┤  │
│      Socket Number        │  │
├───────────────────────────┤  │
│       Enumerator          │  ⎭
├───────────────────────────┤  ⎫
│    Object Field Length    │  │
├───────────────────────────┤  │
│                           │  │
│                           │  │
│         Object            │  │
│                           │  │
│                           │  │
├───────────────────────────┤  │
│     Type Field Length     │  │
├───────────────────────────┤  ⎬  Entity Name
│                           │  │
│                           │  │
│          Type             │  │
│                           │  │
├───────────────────────────┤  │
│     Zone Field Length     │  │
├───────────────────────────┤  │
│                           │  │
│                           │  │
│          Zone             │  │
│                           │  │
└───────────────────────────┘  ⎭
```
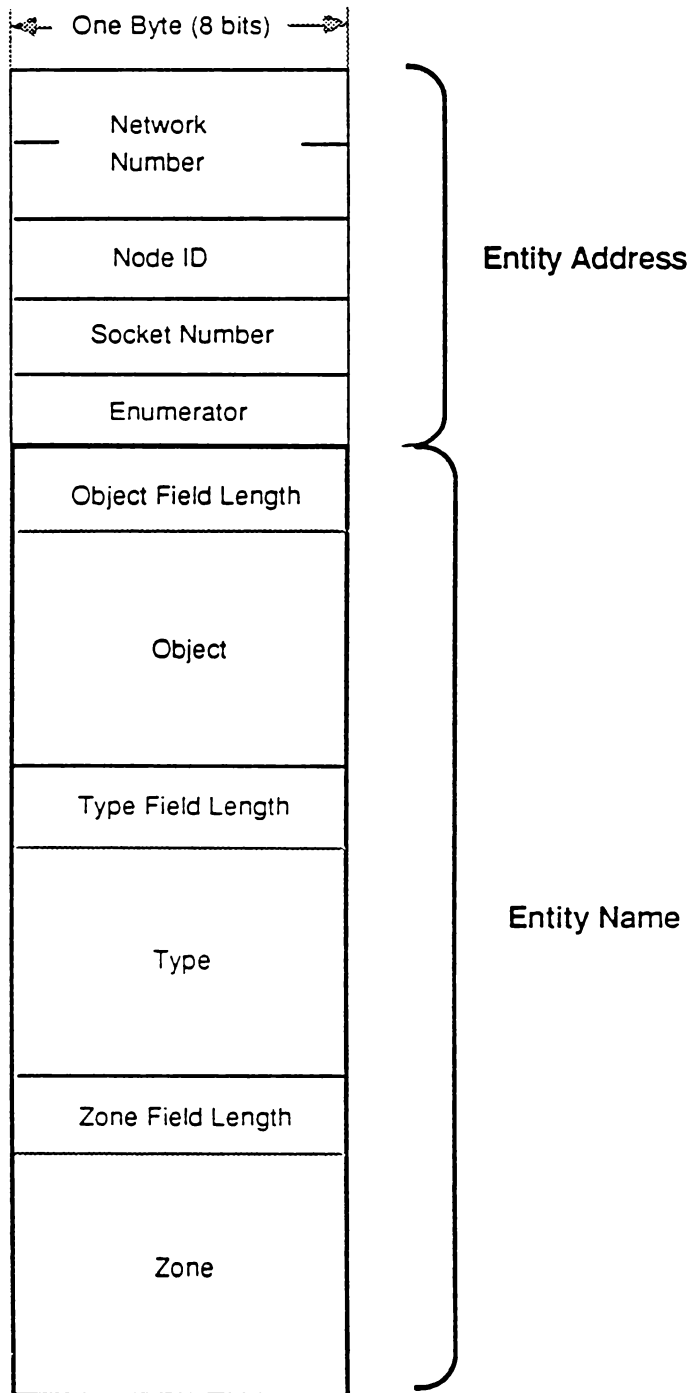
Figure VI-2: NBP Tuple

# VII. AppleTalk Transaction Protocol (ATP)

## About AppleTalk Transaction Protocol

The fundamental purpose of reliable transport protocols is to provide loss-free delivery of client packets from source socket to destination socket. Various features can be added to this basic service, to obtain characteristics appropriate for specific needs. Transport characteristics can be added by clients or built into standard value-added transport services.

The AppleTalk Transaction Protocol (ATP) is a transport protocol that satisfies the transport needs of a large variety of peripheral devices, and the transaction needs for more general networking on AppleTalk. Particular attention has been paid to ease of implementation, so that nodes with tight memory space restrictions will be able to support a sufficient subset of ATP.

## Transactions

Often, a socket client must request the client of another socket to perform a particular higher level function and to report the outcome. This interaction, between a requester and a responder, is termed a *transaction*.

The basic structure of a transaction, in the context of a network, is illustrated in figure VII-1. The requester initiates the transaction by sending a transaction request, TReq, from the requester's socket to the responder's socket. The responder executes the request and returns a transaction response, TResp, reporting the transaction's outcome.

### At-Least-Once Transactions (ALO)

This basic process must be performed in the face of various situations inherent in the loosely-coupled nature of networks, for example:

- TReq is lost in the network,
- TResp is lost or delayed in transit,
- the responder "dies" or becomes unreachable from the requester.

There could be several transaction requests outstanding, and the requester must be able to distinguish between the respective responses received by it over the network. This can be done by using a transaction identifier (TID), sent with the request. The response must contain the same TID as the corresponding request. The TID, in a sense, "binds" together the request and the response portions of a transaction.

In any of the above three error situations, the requester will not receive the transaction's response and must conclude that the transaction was not completed. A recovery procedure must then be activated at the requester. This consists of a timeout and an automatic retry mechanism. If the timer expires in the requester and the response has not been received, the requester retransmits the TReq (see figure VII-1). This process is repeated until a response is received or until a maximum retry count is reached. If the retry count hits its maximum value then it is concluded that the responder is either "dead" or unreachable, and the transaction requester (the ATP client at the requester) is so notified.

This mechanism does its best to ensure that the transaction request is executed at least once. Such a mechanism is adequate if the request is essentially idempotent, i.e. repeated execution of the request is the same as executing it once. (An example of an idempotent transaction is asking a destination station to identify itself.)

Exactly-Once Transactions (XO)

Nevertheless, with this recovery mechanism, lost or delayed responses could cause the transaction to be executed more than once. If the request is not idempotent, serious damage could result. For such requests it is desirable to have a transaction service which ensures transaction execution once and exactly once.

Whether the at-least-once or the exactly-once level of service is appropriate can only be determined by the transaction requester.

The basic technique for implementing an exactly-once transaction protocol is as follows (see figure VII-2). The responder maintains a transactions list of all recently received transactions. Upon receiving a TReq the responder searches through this list to determine if the request has already been received (duplicate transaction-request filtering). A newly received request is inserted into the list and is then executed. After it has been executed, the response is sent back and a copy of the response is attached to the transaction's entry in the transactions list. Upon receiving a duplicate request for which a response has already been sent, the responder retransmits the response. If a duplicate request is received, and a response has not been sent out yet (the request is still being executed), then ATP ignores the duplicate request.

Upon receiving a TResp, the requester should return a transaction release (TRel) packet to release the request from the responder's transactions list. If this TRel gets lost then the request would stay in the list "forever". To eliminate this situation, the responder "timestamps" a request before inserting it in its list. The list is checked periodically and requests that have been in it for too long are eliminated.

This method of filtering duplicate requests by consulting a list of recently received transactions is quite effective in ensuring exactly-once service in most environments. However, it does not guarantee exactly-once service in *all* environments. In fact, if the situation is such that packets are guaranteed, if they arrive at all, to arrive in the order in which they were sent (e.g. on a single AppleTalk network), then this technique is fully effective. However, in an internet environment, on account of multiple paths from source to destination and different transmission delays on these paths, packets may arrive at their destination in an order different from the one in which they were sent. Thus, unusual situations such as the one illustrated in figure VII-3 can arise. Here, the original transaction response was delayed long enough in the internet to provoke a retransmission of the request. Furthermore, the TRel sent by the requester upon receiving the response arrives before the retransmitted request. The responder releases the request from the list upon receiving the TRel, and thus the retransmitted request cannot be filtered out as a duplicate. In fact, with a connectionless protocol it is difficult to guarantee exactly-once service under internet conditions. ATP XO mode does, however, greatly reduce the amount of work a connection-based client must perform to realize true exactly-once service. *ATP XO guarantees that, if a duplicate request is received by the responder, the transaction has already been completed and the request can be ignored.* Clients desiring absolutely guaranteed exactly-once service over an internet should employ some form of simple sequence number checking in addition to ATP to achieve guaranteed internet exactly-once service (an example of this is detailed in the chapter on the Printer Access Protocol).

Note that ATP XO should be considered an optional part of ATP. Nodes which do not require exactly-once service do not need to implement it. It should be kept in mind that higher level protocols, may require ATP XO (for example the Printer Access Protocol and the AppleTalk Session Protocol).

## Multi-Packet Responses

ATP uses the basic idea that a transaction is a request issued by a client in a requesting node to a client in a responding node. The client in the responding node is expected to service the request and generate a response. It is assumed that these clients have some method of unambiguously identifying the data/operation sought in the request (e.g. read a disk block, block number, etc.).

This is a very simple model, in principle sufficient for all interactions. The difficulty is that the underlying network places a size restriction on the packets that can be exchanged. Thus, for instance, the transaction response might not fit in a single packet. For this reason we look upon the transaction request and response as "messages" (not packets). Although ATP restricts its transaction requests to single packets, it allows the transaction response message to be made up of several packets, which of course bear a sequential relationship to one another. When the requesting node receives all the response packets (i.e. the complete response message), the transaction is considered complete and the response is delivered as a single logical entity to the ATP client (the transaction requester).

ATP supports both 'at-least-once' and 'exactly-once' modes as client-elected options.

If the 'at-least-once' case is chosen then ATP handles timeouts and retransmission of requests but does not handle retransmission of responses. In this case, if request actions are not idempotent, it is up to the responding client to handle retransmission of responses to duplicate requests.

The maximum size (number of packets) of a transaction response message is limited to 8 packets. The maximum amount of data in an ATP packet (request or response) is 578 bytes. This limit is derived from the DDP maximum packet size of 586 bytes minus ATP's header size of 8 bytes.

## Transaction IDs

Transaction IDs are generated by the requester and sent along with the TReq packet. An important design issue is the size (16 bits for ATP) of these IDs. This is a function of the rapidity with which transactions are generated and of the maximum packet lifetime (MPL) for the complete network system. The longer the MPL, the larger the TID must be. Similarly, if transactions are generated rapidly, then the TIDs must again be larger. The basic problem is that the TID being of finite size wraps around and there is the danger that for a particular value an old packet stored in some internet router may arrive later on and be accepted as a valid packet.

For a single AppleTalk network, the time taken for exchanging a TReq and a TResp is bounded (by the ALAP) to be greater than about 2.5 msec. Thus there can be no more than 400 transactions per second. From this point of view a single byte TID would allow half a second or so for wraparound of the TID.

However, with network interconnection through store-and-forward internet routers, the impact of MPL (of the order of 30 seconds) makes a one-byte TID inadequate. A 16-bit TID would increase the wrap around time to be approximately two minutes. This obviates concerns about old retransmitted transaction requests and responses "sneaking-in" due to wraparound.

Later in this chapter, the issue of generating transaction identifiers will be revisited to account for another subtle but important characteristic of ATP, namely transactions with infinite retries.

## ATP Bitmap/Sequence Number

Every ATP packet includes in its header a bitmap/sequence number. This field is 8 bits wide. ATP handles lost or out-of-sequence response packets by using this bitmap. The significance of this field depends on the type of ATP packet (TReq, TResp or TRel).

In TReq packets this field is known as the transaction bitmap. The basic idea behind the use of this field is that the requester reserves enough buffers for the expected transaction response, and then sends out the TReq packet indicating to the responder the number of buffers reserved. This is done by setting a bit in the TReq packet's bitmap, for each reserved buffer. The responder can then examine the TReq packet's bitmap and determine the number of packets the requester is expecting to receive in the transaction response message.

In TResp packets this field is known as the ATP sequence number. The value of this field in the TResp packet is an integer (in the range 0 to 7), indicating the sequential position of that packet in the transaction response message. The requester can use this value to put the received response packet in the appropriate response buffer (even if the response packet is received "out-of-sequence") for delivery to the transaction requester (ATP client). Furthermore, the requester clears the corresponding bit in its copy of the transaction bitmap.

The actual transaction response message may turn out to be smaller than was expected by the requester. Thus a provision is made in the response packet's header to signal "end-of-message" in the last response packet when it is sent out by the responder. Upon receiving a response packet with the end-of-message indication, the requester must clear all bits in its copy of the transaction bitmap corresponding to higher sequential positions. Note that this end-of-message signal is internal to ATP; the responding client tells ATP to set it, but it is not necessarily delivered to the requesting client and should not be used for higher-level communications (e.g. as an end-of-file indicator).

If the requester's retry timeout expires and the complete transaction response has not been received as yet (indicated by one or more bits still set to one in the requester's transaction bitmap), then a TReq is sent out again with the current value of the transaction bitmap and the same TID. Thus only the missing transaction response packets are requested again.

This mechanism is illustrated in figure VII-5 where a requester issues a TReq indicating that it has reserved six buffers for the response. For instance, the request might be for six blocks of information from a disk device. The TReq packet would have in its ATP data part the pertinent information: what file, which six blocks, etc. ATP builds the request packet and sets the least significant six bits in the bitmap. When the responder receives this request packet, it examines the request's ATP data and its bitmap and thus determines the type and range of the request to be serviced. The six blocks are fetched from disk, passed

to the ATP layer in the device node and sent back to the requesting node, each in a separate packet with its sequence number indicating the position in the response.

The example illustrated assumes that the third response packet is lost in the network. Thus the retry timeout will expire in the requester, which then retransmits the original request (transparently to the ATP requesting client) but with a bitmap reflecting only the missing third response packet.

Note that single packet request-response transactions are simply the degenerate case in which the transaction request has only one bit set in its bitmap. If two nodes wish to communicate in this manner, very little extra packet overhead is added by the protocol.

## Responders with Limited Buffer Space

A potential difficulty with exactly-once transactions is that a responder might not have enough buffer space to hold the entire transaction response message until the end of the transaction (i.e. receipt of a TRel).

ATP provides a mechanism for such responders to reuse their buffers, through a confirmation of response packet delivery. This is done by piggy-backing, in a response packet, a request to Send Transaction Status (STS). The requester, upon receiving such a response packet, immediately sends out a TReq with the current bitmap (i.e. indicating which response packets have still not been received and hence providing a way to determine which have already been received). The responder can then use this bitmap to free buffers holding already-delivered response packets.

Two user-interface issues arise in connection with the STS bit. The retransmitted TReq would normally be filtered by ATP XO as a duplicate, so ATP must provide some way of delivering the updated bitmap to the user without delivering the duplicate request. Related to this is the fact that, in an internet, TReq's can get out of order; if a duplicate TReq is ever received whose bitmap indicates that fewer responses have been received than a previous TReq indicated, it should be ignored as a delayed duplicate and should not be passed to the user.

Figure VII-6 illustrates the use of STS with an example in which the responder, with two buffers, services a request for a seven packet response.

TReq packets sent in response to an STS do not consume the retry count, but do reset the Retry TimeOut.

## ATP Packet Format

The format of an ATP packet is illustrated in figure VII-7. It consists of an 8-byte ATP header plus up to 578 ATP data bytes.

The first byte of the ATP header is used for command/control information (CCI). The two high-order bits of the CCI are the packet's function code. These two bits are encoded as follows:

> 01 -- TReq packet
> 10 -- TResp packet
> 11 -- TRel packet.

The EOM bit is set in a TResp packet to signal that it is the last packet in the transaction's response message. The XO bit must be set in all TReq packets that pertain to the exactly-once mode of operation of the protocol. The STS bit is set in TResp packets to force the requester to retransmit TReq immediately. The remaining three bits of CCI must always be zero.

The 8 bits immediately following the command/control field contain the ATP bitmap/sequence number. The packets comprising the transaction response message are assigned sequence numbers 0 through 7. The sequence number (encoded as an integer) is sent in the ATP bitmap/sequence number field of the corresponding response packet.

In the case of a transaction request packet, a bit of the bitmap is set to one for each expected response packet. The least significant bit corresponds to the response packet with sequence number 0, up through the most significant bit which corresponds to sequence number 7.

The third and fourth bytes of the ATP header contain the 16-bit transaction ID. TIDs are generated in the ATP requester, and are incremented from transaction to transaction as unsigned 16 bit integers (the value zero is permitted).

The last four bytes of the ATP header are not examined by ATP, but contain user data. As such, they should, strictly speaking, not be considered part of the ATP header. However, they can be used by an ATP client to build a simple header for a higher level protocol. The reason they have been separated out is to allow an implementation of ATP that handles the complete ATP response message's data in an assembled, contiguous, form, without interposed higher level headers. ATP-Client interfaces must build appropriate mechanisms for exchanging these four *user bytes* independently of the data. It should also be noted that the ATP user bytes contained in a TRel packet are not significant and clients should not use them in any way.


## ATP Interface

The interface to ATP is made up of the five calls described below. It is convenient to visualize the ATP package (an implementation of ATP) as consisting of two parts: the ATP Requester and the ATP Responder. The calls are discussed in the context of each end.

It is not appropriate in this specification of the protocol to detail the interface, as several aspects are implementation dependent. The description below is in general terms, adequate for establishing the characteristics of the ATP service to the next higher layer.

Remember that we neither assume nor require the availability of a multiprocessing environment in the network node. Our descriptions of the various interface calls have been written in a generic form indicating parameters passed by the caller to the ATP implementation, and results and outcome codes returned by the latter. The result codes and their interpretation depend on the specifics of the implementation of a call. If the call is issued synchronously, the caller is blocked until the call's operation has been completed or aborted, then the returned parameters become available when the caller is unblocked. In the case of asynchronous calls, a call completion mechanism is activated when the operation completes or aborts. Then the returned parameters become available through the completion routine mechanism.

We envision at least two kinds of interfaces: a packet-by-packet passing of response buffers to and from ATP, and a response message (i.e. in a contiguous buffer) interface.

These are analogous to the familiar packet stream and byte stream interfaces available for data stream protocols. However implementors are completely free to provide any type of interface they consider appropriate.

## Send a Request

The transaction requester (ATP client) issues this call to send a transaction request. It must supply several parameters with the call. These include the address of the destination socket, the ATP data part and user bytes of the request packet, buffer space for the expected response packets, and whether the exactly-once mode of service is required or not. In addition the caller specifies the duration of the retry timeout to be used and the maximum number of retries. There must be a provision in the interface for the caller to indicate "infinite retries," i.e. keep trying the request until a response is obtained. It may also be desirable for the caller to be able to optionally specify the socket through which the request should be sent (or ATP could pick one for him).

• Call Parameters:

>   transaction mode (exactly-once or at-least-once)
>   transaction responder's address (network number, node ID and socket number)
>   ATP request packet's data part and its length
>   ATP user bytes
>   expected number of response packets
>   buffer space for the transaction response message
>   retry timeout in seconds
>   maximum number of retries
>   [optional] socket through which to send the request

• Returned Parameters:

>   result code:  (success, failure)
>   number of response packets received
>   user bytes from responses

An outcome code of failure is returned if ATP has exhausted all retries and a complete response has not been received. Note that no error is returned if the caller requested exactly-once service but the responder does not support it -- the request will be executed at least once.

An outcome code of success is returned whenever a complete response message has been received. A complete response is said to have been received if either of the following occurs: (i) all response packets originally requested have been received, or (ii) all response packets with sequence number 0 to some integer n have been received and packet n had the EOM bit set.

In either case, the actual number of response packets received is always returned to the requesting client. A count of zero should indicate that the other end did not respond at all. In the case of a nonzero count, the client can examine the response buffers to determine which portions of the response message were actually received and to detect missing pieces for higher level recovery, if so desired.

Open a Responding Socket

An ATP client uses this call to instruct ATP to open a socket (well-known or dynamically assigned) for the purpose of receiving transaction requests. If well-known, the client passes the socket number to ATP; otherwise the dynamically assigned socket number is returned to the client.

When opening this socket, the client is in effect opening a "transaction listening socket". The call allows the socket to be setup so that requests are accepted from a specified network address (provided in the call). This address can include a zero in the network number, node ID, or socket number fields to indicate that any value is acceptable for that field.

• Call Parameters:

> transaction listening socket number (if well-known)
> admissible transaction requester address (network number, node ID, and
> socket number)

• Returned Parameters:

> result code (success, failure)
> local socket number (if dynamically assigned)

Note that this call does not set up any buffers for the reception of transaction requests. That is done by issuing the Receive-a-Request call.

Close a Responding Socket

This call is used to close a socket previously opened with the Open-a-Responding-Socket call.

• Call Parameters:

> transaction listening socket number

• Returned Parameters:

> result code (success, failure)

Receive a Request

The transaction responder issues this call to set up the mechanism for actual reception of a transaction request through an already-opened transaction responding socket.

• Call Parameters:

> local socket number on which to listen
> buffer for receiving the request

• Returned Parameters:

> the received request's ATP data
> received request's user bytes
> transaction ID

transaction Requester's Address (network number, node ID and socket number)
bitmap
XO indication

## Send a Response

When a transaction responder has finished servicing the request, it issues a Send-a-Response call to send out one or more response packets. ATP will send out each response buffer with the indicated transaction ID and a sequence number indicating the position of the particular response packet in the response message. There are many different ways of implementing this call; our description is generic.

• Call Parameters:

local socket number (the responding socket)
transaction ID
destination internet address (Network number, node ID and socket number)
transaction response message/packets (ATP data part)
transaction user bytes (up to eight sets)
descriptors to determine the sequence numbers of the response packets
EOM and STS control

• Returned Parameters:

result code:   (success, failure)

## ATP State Model

The following description provides a sufficiently precise statement of the protocol internals for the purpose of implementing the protocol. It is not a formal specification, but an aid for protocol implementers.

The description is presented in terms of the actions to be taken in response to all possible events. Certain special terms are employed in the description, and we start with a discussion of some of these.

First of all, the ATP requester must maintain all information necessary for retransmitting an ATP request and for receiving its responses. This is referred to by us as the Transaction Control Block (TCB). More specifically, this would contain all the information provided by the transaction requester in the Send-a-Request call, plus the TID, the request's bitmap and a response-packets-received counter. With each request and thus with each TCB we associate a retry timer, used to retransmit the request packet in order to recover from loss of request or response packet situations.

In the second place, the ATP responder must maintain a Request Control Block (RqCB) for each Receive-a-Request call issued by a client in that node. This block contains the information provided by that call including all pertinent data as to the buffers and delivery-to-client mechanism (implementation dependent).

A third data structure, the Response Control Block (RspCB) is needed only in nodes implementing the exactly-once mode of operation. It holds the information needed to filter duplicate requests and to retransmit response packets in response to these duplicates. We associate a release timer with each RspCB. This timer is used to release the RspCB in the

event that the release packet sent by the requester is lost. The transactions list mentioned earlier consists of these RspCBs.

The release timer is specified to be 30 seconds long. Note that the release timer is started as soon as a RspCB is set up (i.e. upon receiving the TReq). It is reset every time a corresponding TResp is sent out. This implies that the responding client must send out the first TResp within a thirty second interval of the TReq's arrival, and subsequent TResps at a maximum separation of thirty seconds from each other. Failure to do so will result in the RspCB being destroyed and a duplicate request being delivered to the responding client.

## ATP Requester

*Send-a-Request call issued by a transaction requester in the node:*

      a. validate the following call parameters:
           - number of response packets should be at most 8
           - the ATP request's data should be at most 578 bytes long;
      if either parameter is invalid, then reject the call;

      b. create a TCB:
           - insert the call parameters in it
           - clear the response-packets-received counter
           - insert the Retry Count into the TCB;

      c. generate a TID:
           - this TID must be such that the packets of the new transaction will be
             correctly distinguished from those of other transactions; details of TID
             generation are discussed at the end of this chapter;
      save the TID in the TCB;

      d. generate the bitmap for the request packet and save it in the TCB;

      e. prepare the ATP header:
           - insert the TID and the bitmap
           - set the function code bits to binary 01
           - {only for systems implementing the exactly-once mode of operation}:
             if the caller requested exactly once mode the set the XO bit;

      f. call DDP to send the ATP request packet (disregard any error returned);

      g. start the request's retry timer.

*Retry timer expires:*

      a. if Retry Count = 0 then:
           - set outcome code to 'failure'
           - notify transaction requester (the client in the node) of the outcome
           - destroy the TCB;

      b. if Retry Count <> 0 then:
           - decrement the retry count (if not "infinite")
           - change the bitmap in the ATP request's header to current value in the TCB
           - call DDP to retransmit the request packet (ignore errors)

- start the retry timer.

*ATP Response Packet received from the network (i.e. from DDP):*

    a. use the packet's TID and source address to search for the TCB;

    b. if a matching TCB is not found then ignore the packet and exit;

    c. if a matching TCB is found then check the packet's sequence number against the TCB's bitmap to determine if this response packet is expected. The packet is expected if the bit corresponding to the reponse packet's sequence number is set in the TCB's bitmap. If the packet is not expected then ignore it and exit;

    d. if the response is expected then:
        - clear the corresponding bit in the TCB bitmap
        - set up response packet's ATP data for delivery to transaction requester
        - increment the response packets counter in the TCB;

    e. if the packet's EOM bit is set then clear all higher bits in the TCB bitmap;

    f. if the packet's STS bit is set then:
        - call DDP to re-send the request with the current TCB information
        - reset the retry timer for the request;

    g. if the TCB bitmap = 0 then: {a complete response has been received}
        - cancel the retry counter
        - set the outcome code to 'success'
        - {only if exactly-once mode is implemented}
            if the transaction is of exactly-once mode (determined by examining the TCB) then call DDP to send a TRel packet to the responder
        - notify the transaction requester
        - destroy the TCB.

## <u>ATP Responder</u>

*Open-a-Responding-Socket call issued by a Transaction Responder in the Node:*

    a. if caller specifies a well-known socket then call DDP to open that socket else call DDP to open a dynamically assigned socket;

    b. if DDP returns with error then set result code to the error;

    c. if DDP returns without error then
        - set result code to 'success'
        - save socket number and the admissible transaction requester address in an ATP responding sockets table;

    d. return to caller .

*Close-a-Responding-Socket call issued by a transaction responder in the Node:*

    a. call DDP to close the socket;

   b. release all RqCBs, and, for systems supporting exactly-once mode
      of operation, release all RspCBs (and cancel all release timers), if any,
      associated with the socket;

   c. delete the socket from the ATP responding sockets table.

*Receive-a-Request Call issued by the transaction responder in the node:*

   a. if the specified local socket is not active then return to caller with error;

   b. create a RqCB and attach it to the socket;

   c. save the call's parameters in the RqCB.

*Send-a-Response call issued by transaction responder in the node:*

   a. if the local socket is invalid OR response data lengths are invalid then
      return to caller with error;

   b. (only if exactly-once mode is implemented)  Search for a RspCB matching
      the call's local socket number, TID, and transaction requester address
      (destination of the ATP Response).  If found then save the response
      attached to the RspCB (for potential retransmission in response to
      duplicate requests received subsequently), and restart release timer;

   c. send the response packets through DDP, setting the ATP header of each with
      function code binary 10, with the caller supplied TID, with the correct sequence
      number for the packet's sequential position in the response message, with the
      EOM flag set in the last response packet (and the STS flag if requested).
      Ignore any error returned by DDP.

*Release timer expires:*  (only if exactly-once mode of operation is implemented)

   a. destroy the RspCB and release all associated data structures.

*ATP Request Packet (TReq) received from DDP:*

   a. (only if exactly-once mode is implemented) If the packet has its XO bit set and a
      matching RspCB (i.e. the packet's source and destination addresses are the same
      as those saved in the RspCB and the packet's TID is equal to the one saved in the
      RspCB) exists then:
         - retransmit all response packets
         - restart the release timer
         - return the bitmap to the client if a previous response had the STS bit set
         - exit;

   b. if a RqCB does not exist for the local socket or if the packet's source
      address does not match the admissible requester address in the RqCB then
      ignore the packet and exit;

   c. (only if exactly-once mode is implemented) If packet's XO bit is set then create a
      RspCB (save therein the request's source and destination addresses and its
      transaction ID) and start its release timer;

d. notify the client about the arrival of the request and destroy the corresponding RqCB.

*ATP Release Packet (TRel) received from DDP*: {only if exactly-once mode is implemented}

a. search for a RspCB that matches the packet's TID and source and destination addresses. If not found then ignore the release packet;

b. if a matching RspCB is found then:
- destroy the RspCB and all release associated data structures
- cancel the RspCB's release timer.

## Some Optional ATP Interface Calls

In certain instances the clients of ATP might use certain contextual information to enhance their use of ATP through some additional interface calls. Here are two examples. These calls have been found useful in implementing certain higher level protocols, but are completely optional for a given ATP implementation.

### Release a RspCB

In the first case two clients of ATP are communicating with each other, using the exactly once mode, and have decided to have at most one outstanding transaction at a time. Client A calls ATP to send a TReq packet to client B. B sends back the response. A, upon receiving the response, sends out a second request (but no release packet). The second request packet upon being received by B signals that the response to the previous request has been received by A. Now B could simply call its ATP responder and ask it to release the previous transaction's response control block. This needs a Release-RspCB call to the ATP responder. The parameters of this call are fairly obvious in this instance.

### Release a TCB

Another instance arises when a client A wishes to send data to client B. A must first inform B of this intention, and thus allow B to request the data. To this end, A can send a TReq to B to signal "I want to write N bytes of data to you, please ask me for it on my socket number S". Instead of sending a TResp to this packet, B could just send a TReq to A's socket S asking for the data. The reception by A, on socket S, of B's request implies that A's original request has been received by B. A could call its ATP requester and ask it to eliminate the previous transaction's TCB. This needs a Release-TCB call to the ATP requester. This call could also be used by the requesting-end client to cancel an outstanding ATP request at any time -- for instance to abort an infinite retry request.

Details of these calls are implementation-dependent and do not affect the ATP protocol per se. We mention these as interesting variants that might be implemented by certain clients who wish to reduce the traffic generated on the network.

## Wrap Around and Generation of Transaction Identifiers

Earlier in this chapter it was noted that transaction IDs being of finite size wrap around and that there is the danger that for a particular value an old packet stored in some internet router

may arrive later and be accepted as a valid packet in a later transaction using the same identifier. On the basis of a maximum packet lifetime estimate of 30 seconds, it was determined that this problem could be avoided if the TID were 16 bits long (wrap around is estimated to take approximately two minutes).

There is yet another important problem of this nature that stems from the fact that a particular transaction might take much more than two minutes to complete. For instance, if the request is to search through an "encyclopedia" for all references to a particular piece of information, this might take a very long time indeed (several minutes). In this case, the transaction ID could wrap around and another transaction is then issued with the same TID, leading to the same dilemma as the one mentioned in the previous paragraph. ATP does not prohibit operations of this sort. In fact, it is because such transactions might be made, that the specification of the length of the ATP retry timer and maximum retry count is left up to ATP's transaction requesting client.

In the same vein, ATP allows the requester to issue an ATP transaction with maximum retry count set to "infinite". In this case, ATP continues to retransmit the transaction request until a reply is received. If a reply is not sent, then the transaction will be retransmitted infinitely often. This leads to the same TID wrap around problem. In fact, some protocols like the AppleTalk Session Protocol use infinite-retry transactions for the purpose of tickling the other end of a session (i.e. informing it that this end is alive). These "tickle" transactions never receive a reply.

A properly implemented ATP will function correctly in the face of these wrap around scenarios. There are two aspects to this: the use of TIDs to distinguish between transactions and the generation of TIDs.

The ATP requesting end generates the TID when it is asked by a client to send a transaction request. At that time, it creates a TCB and saves several pieces of information in it. These include the number of the local socket through which the transaction is being sent, the complete internet address of the responding socket to which it is being sent, and the just generated TID. This information is saved in order to ensure a correct match of the response packets with the transaction. When the ATP requesting end receives a transaction response it identifies the corresponding request by looking for a TCB whose saved information matches the response packet's TID and the packet's source and destination socket addresses.

Thus TID wrap around by itself is not dangerous except if it causes the simultaneous existence of two or more transactions (i.e. TCBs) with the same TID and the same requesting and responding socket addresses. This observation allows the specification of an algorithm for generating TIDs that ensures that wrap around has no ill effects. This is as follows:

```
{ Algorithm used by ATP Requesting end to generate TID for a new transaction }
new_TID := last_used_TID;
Not_In_Use := TRUE;
REPEAT
     new_TID := (new_TID + 1) modulo 2^16;
     Search all TCBs on the local requesting socket, and if any one of these has
          (new_TID = TCB's TID) then set Not_In_Use := FALSE;
UNTIL Not_In_Use;
{ At this point new_TID has the newly generated transaction identifier }
last_used_TID := new_TID;
```

Note that this algorithm ignores the TCB's destination socket address (i.e. does not compare it with the address of the new transaction's destination). This simplification of the algorithm does not however in any way reduce its effectiveness in preventing the wrap around problem.

Requesting
End

Responding
End

Sends transaction
request

TReq(TID)

Executes
the request

Sends transaction
response

TResp(TID)

time

time

Requesting
End

Responding
End

TReq(TID)

TResp(TID)

Retry
Timeout

X Lost

TReq(TID)

TResp(TID)

time

time
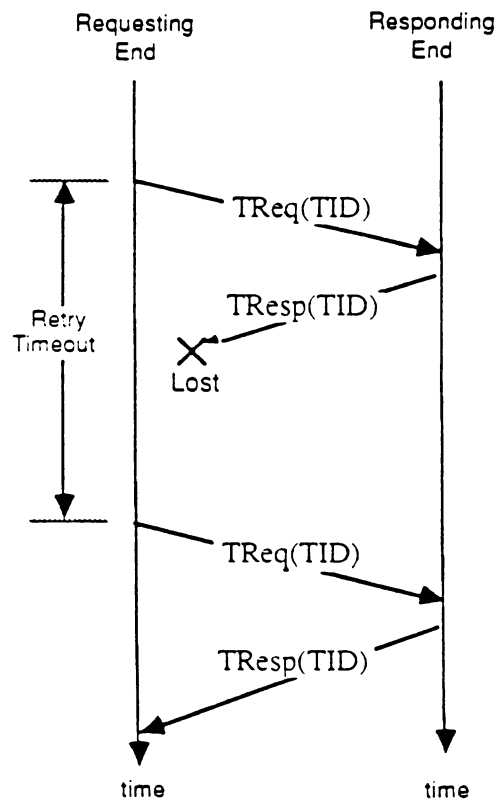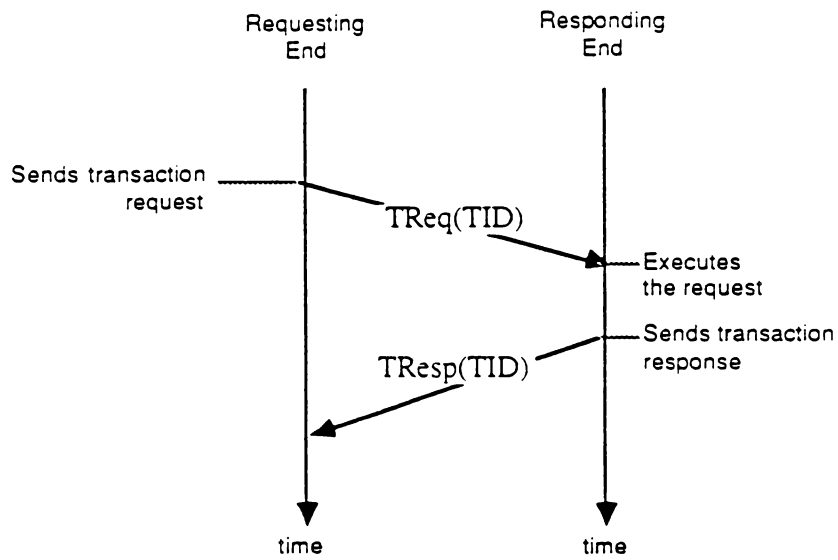
Figure VII-1:  Transaction Terminology
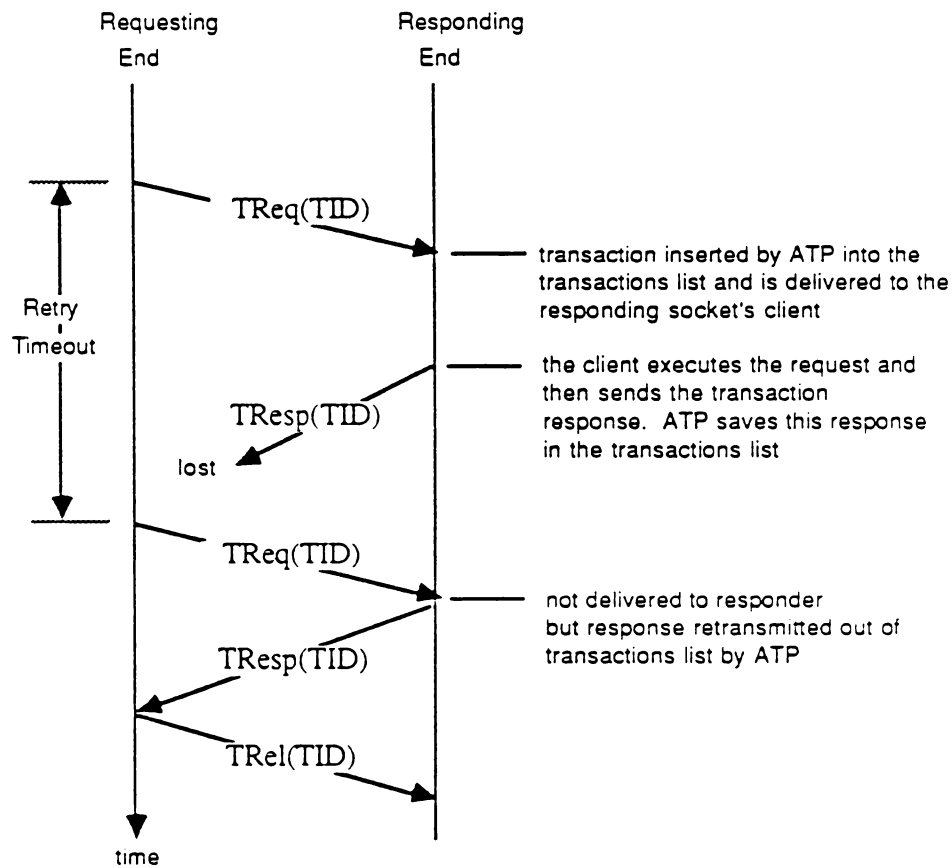
Figure VII-2: Exactly-Once Transactions

Requesting
End

Responding
End

Retry
Timeout

TReq

Request is
retransmitted

TResp

Response arrives
Request completed
TRel sent

TRel

TReq arrives before
retransmitted request.
release transaction
from transactions list

TReq

Retransmitted request arrives
Cannot be filtered as duplicate
Passed on to client

time

time
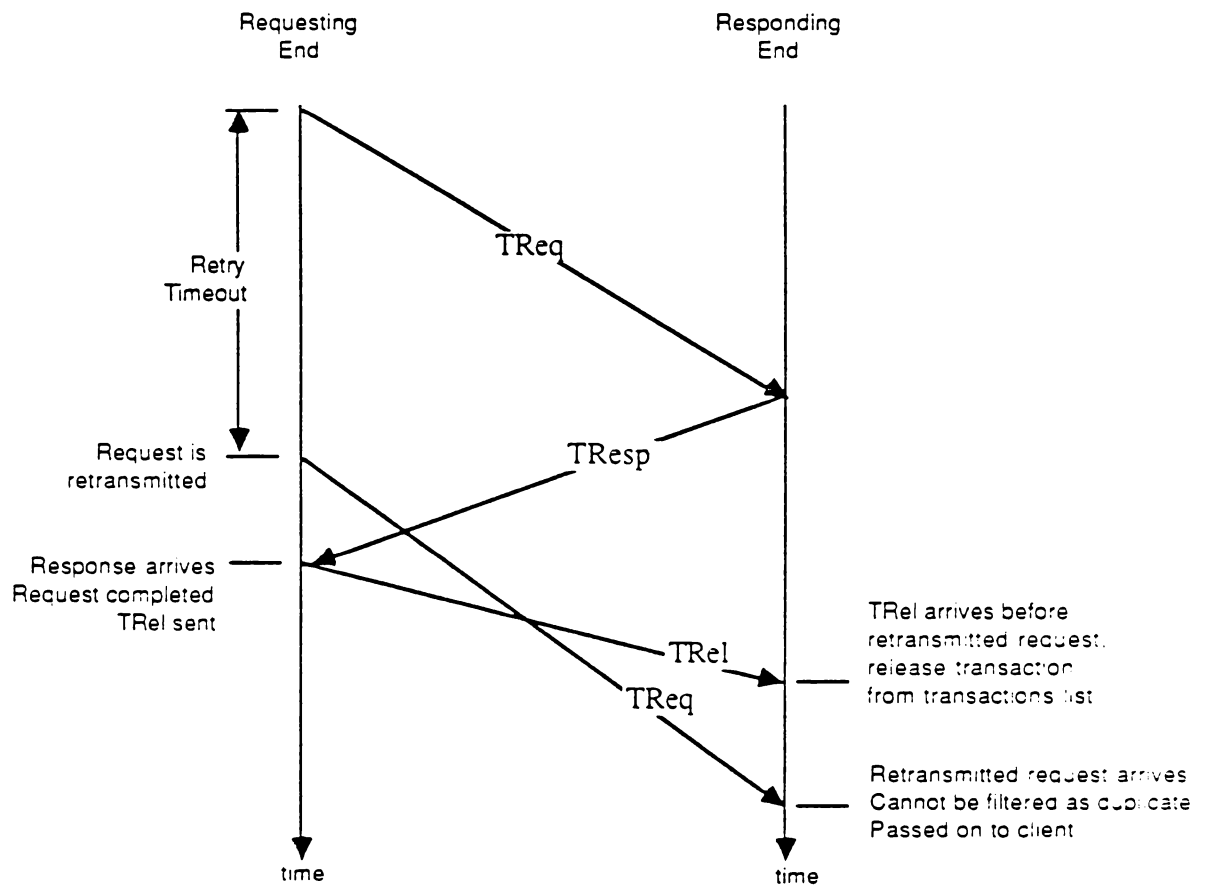
Figure VII-3:  Duplicate delivery of Request in Exactly-Once Mode
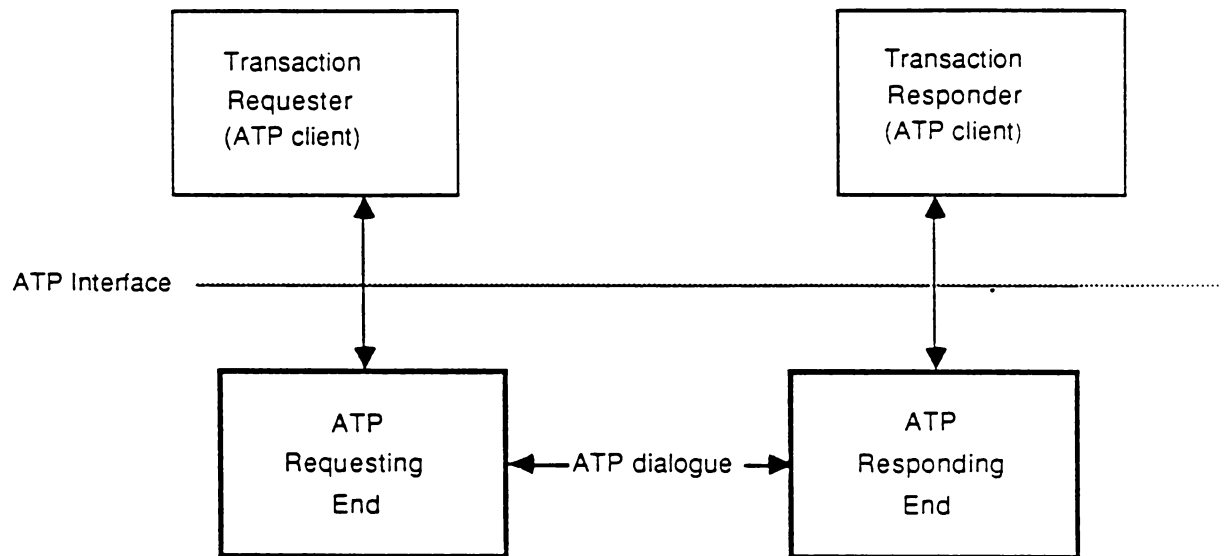
**Figure VII-4: ATP Terminology**

Requesting
End

Responding
End

TReq
(bitmap = 00111111)

Retry
Timeout

TResp(0)

TResp(1)

TResp(2)

X

.
.
.

TResp(5)

TReq
(bitmap = 00000100)

TResp(2)

time

time
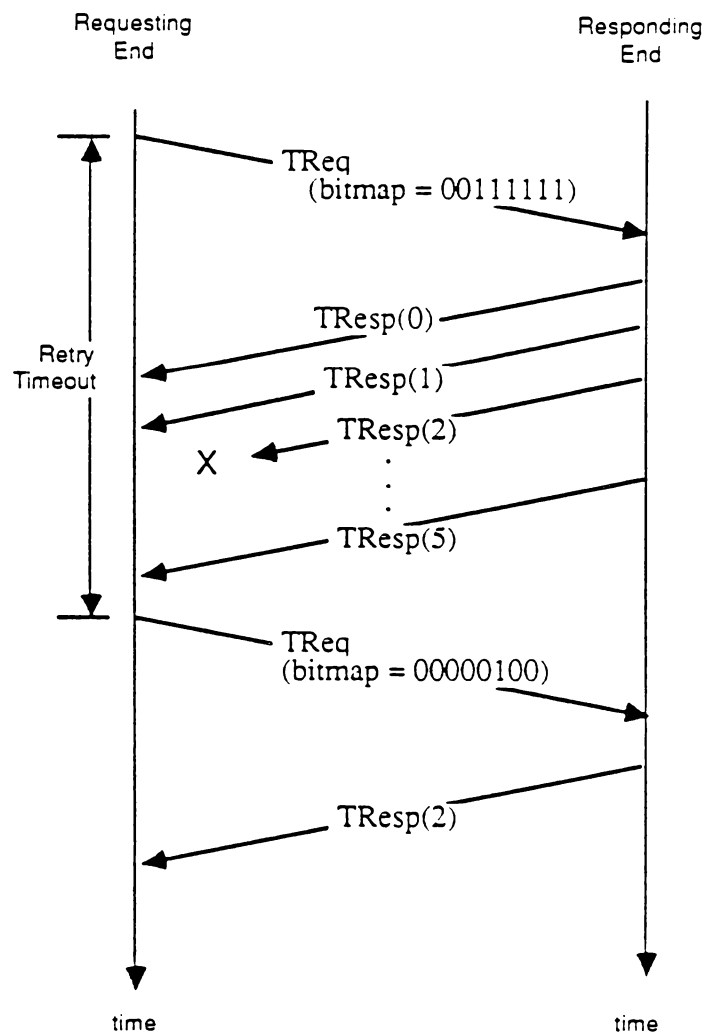
Figure VII-5: Multiple-Packet Response

Requesting
End

Responding
End
(has only two buffers)

TReq
(bitmap = 01111111)

in buffer A

TResp(0)

in buffer B

Lost

TResp(1) + STS

TReq
(bitmap = 01111101)

frees buffer B

in buffer A

TResp(0)

in buffer B

TResp(2) + STS

TReq
(bitmap = 01111000)

frees buffers
A and B

in buffer A

TResp(3)

in buffer B

TResp(4) + STS

•

•

•

time

time

Figure VII-6:  Use of STS

One Byte (8 bits)

LAP Header

DDP Header

Command/Control field

| | | | | | 0 | 0 | 0 |

STS bit

EOM bit

XO bit

function
code

DDP Protocol type = 3

Bitmap/
Sequence No

hi bits

lo bits

Transaction
Identifier

ATP Header

4 User Bytes

ATP Data

(0 to 578. bytes)

Figure VII-7:  ATP Packet Format

# VIII. Zone Information Protocol (ZIP)

## About Zone Information Protocol

The Name Binding Protocol introduced the concept of a zone, defining a zone as an arbitrary subset of networks in the internet in which name lookups were performed. A given network was said to be in one and only one zone. It was specified in that chapter that the bridges were responsible for maintaining the mapping between networks and zone names, and for converting nodes' *broadcast requests* (BrRq's) into zone-wide lookups. This network-to-zone-name mapping is maintained by the Zone Information Protocol (ZIP). This chapter describes that protocol and its uses.

## ZIP Services

An important feature of the Zone Information Protocol is the fact that most of its services are transparent to the normal (non-bridge) node, as they are imbeded in the use of the zone field of NBP. Unless a normal node wishes to be aware of the details of the zone structure of the internet, it does not have to implement a ZIP process in any form. Those normal nodes desiring this information may implement a small subset of the ZIP in their machines. The majority of ZIP, however, is implemented in the bridges. ZIP provides two major services: (1) the maintanence of the network-number-to-zone-name mapping of the internet, and (2) support for the various housekeeping commands which normal nodes may wish to implement for obtaining and possibly changing this mapping. The first portion of ZIP pertains only to bridges; the second concerns both bridges and normal nodes.

## The Network-Number to Zone-Name Mapping

### The Zone Information Table

Under stable conditions, each bridge maintains a complete network number to zone name mapping of the internet, known as the *Zone Information Table*, or ZIT. This table consists of one entry for each network in the internet. A ZIT entry is a tuple of the form <network number, zone name>. The zone name field can be NIL, indicating that the zone name of that network is currently unknown (this is a temporary condition); otherwise it is a (case insensitive) string specifying the zone name of that network.

The ZIP process in a bridge learns about new networks on the internet by monitoring RTMP's routing table. When it discovers an entry in the routing table which is not in the ZIT, it creates a new ZIT entry for that network with a zone name of NIL and initiates an attempt to discover its zone name. Likewise if the ZIP process discovers that the ZIT contains an entry whose network number is not in the routing table, it concludes that that network is no longer on the internet and removes its entry from the ZIT.

### The Zone Information Socket: ZIP Queries and Replies

Associated with ZIP implementations in bridges is a statically assigned socket known as the *Zones Information Socket* or ZIS. This socket, socket 6, is opened by ZIP during its initialization. It is the socket to which other bridges address requests for zone information and through which a given bridge responds to those requests. These requests, known as *ZIP queries*, contain a list of network numbers that the requesting node is attempting to

determine the zone names of. A bridge receiving such a request responds with a *ZIP reply* listing the requested zone names of which it is aware (it does not respond if it is not aware of any of the requested zone names).

### ZIT Maintenance

ZIP requires an additional field in the port descriptor of each bridge port: this is the zone name of that port's network. This field is only necessary for ports connected to AppleTalk networks, and furthermore only for those ports for which the bridge contains seed information. It is valid for this field to be NIL, indicating "unknown", as long as the restrictions specified under "Zone Name Assignment" are adhered to.

Upon being turned on, each bridge's ZIT consists of one entry for each of its directly-connected AppleTalks for which it is a seed bridge. Both the network number and zone name fields of these entries are taken from the port descriptor for those networks. Additionally, ZIP monitors the routing table for the addition of any networks which are not in the ZIT (i.e. networks of which the bridge has just become aware). When it discovers such a network, it creates a new ZIT entry, with a zone name field of NIL. Whenever a ZIT entry is created with a zone name of NIL (either as just described, or at initialization time through being taken from a port descriptor), ZIP sends out a zone query in an attempt to discover the zone name of that network. This query is sent to the ZIS in the node specified as follows: the routing table entry for the given network is examined. If the next bridge field is zero (indicating the network is a directly-connected one), the query is broadcast; otherwise it is sent to the indicated bridge. The next bridge field specifies, in some sense, the start of a route to the desired network. The zone information for a given network can be viewed as propagating outward from that network. In contrast to RTMP however, the information is proactively requested as opposed to reactively received.

Bridges receiving a ZIP query asking for a zone name they are aware of respond (to the requesting socket) with a ZIP reply indicating that zone name. Bridges unaware of this information do not respond at all. As responses are received, the requesting bridge enters the specified zone names into the appropriate ZIT entries. For some queries, no reply may be received (the query may have been lost or the queried bridge may not yet have the information). ZIP does nothing in this case. ZIP does, however, maintain a background timer for the purpose of retransmitting these requests. Whenever this timer expires, ZIP retransmits one query for each entry in its ZIT whose zone name is still NIL. This query is transmitted in exactly the same manner as it was sent originally. In this way, zone names will propagate dynamically outward from the named network itself -- those bridges one hop away will receive the information on their first query, those two hops away may not received it until their second, etc. Eventually on a stable internet, the ZIT in every bridge will contain the entire network to zone name mapping, and there will be no further ZIP activity.

ZIP also monitors the routing table to determine if a network goes down (i.e. a network listed in the ZIT is not in the routing table). In this case ZIP deletes the corresponding ZIT entry (the network may come back up later with a different zone name, which must be re-discovered).

## Changing Zone Names

Under stable conditions, each network's zone name is replicated in the ZIT of each bridge on the internet. Changing a network's zone name requires making sure that that piece of information is changed in every bridge. Somehow each bridge must be notified of the

change, since on a stable internet bridges are no longer sending out zone queries. One possible notification method would be an internet-wide broadcast of the change request. Internet-wide broadcasting, however, is not supported by DDP, and is both complicated and expensive (in terms of network traffic) to implement. Thus ZIP uses an alternate method.

The method used by ZIP for changing the zone name of a given network is conceptually a simple one. ZIP combines with RTMP to logically "bring down" the network. Once the network is down, its entry disappears from all the routing tables in all the bridges on the internet. As the network disappears from all the routing tables, ZIP also removes it from all the ZITs. Once this process is complete, ZIP combines with RTMP to bring the network back up with the new zone name. As a new network coming up, its zone name then propagates out into the internet through the normal zone query process. The network becomes known by the new zone name.

An important aspect of the zone name changing processes it that the network is only logically "down." It does not have to be brought down physically (although it can be if desired). More importantly, ZIP and RTMP combine in such a way as to enable internet traffic to still be routed *through* the "down" network. Although no traffic can be routed *to* the network, all other internet traffic flows normally through it. Thus even a network through which critical internet traffic flows can have its zone name changed with no effect on that traffic.

As alluded to in the RTMP chapter, ZIP requires an additional service of RTMP to enable it to change the zone name of a network: ZIP must be able to cause RTMP to remove an entry for one of its directly connected networks from its routing table, and to re-insert that entry at a later point. Given this service, ZIP can perform the name change. Details follow.

The request to change a network's zone name must originate from a node on that network. The node broadcasts a special *ZIP takedown* request to the ZIS. As a broadcast, it should be sent several times. All bridges on the network receive this request through their ZIS. Upon receiving the request, the ZIP process in each of these bridges instructs its associated RTMP process to delete the routing table entry associated with the network through which the takedown request is received. Since the routing information for a given network originates in and is "refreshed" by the bridges connected to that network, removing this information from all those bridges effectively brings the network down. Through the operation of RTMP, the entry for that network will disappear from the routing table of every bridge on the internet. The network is now "down." Bridges connected to that net can still forward packets between themselves, however, and thus packets can be forwarded through the net.

Once the network is completely down, it can be brought back up with a new zone name. This is done by broadcasting a *ZIP bringup* request to the ZIS on the net which is to be brought back up. This request specifies the new zone name the network is to be brought up with. The ZIP process in all bridges on the network receives the request and instructs its associated RTMP process to recreate the routing table entry for that network. The ZIP process also creates a ZIT entry for that network with the specified zone name. The network is now back up, with the new zone name. The fact that it is up propagates out through the internet via RTMP, and the new zone name propagates outward through the ZIP query process.

A network's zone name can of course also be changed by physically isolating the network from the internet, re-initializing all the bridges with the new zone name, and returning the network to the internet. However, just as in the takedown/bringup case, the network can

not be returned to the internet (brought back up) until information about it has disappeared from all the ZITs in the internet. This is an important point: in either case, a network can not be brought back up with a new zone name for a given amount of time after it is brought down. Although this parameter is somewhat a function of internet topology, ZIP defines it as a constant known as the *ZIP bringback time*. The exact value of the ZIP bringback time is defined in the section on parameters.

In general, non-bridge nodes need not be concerned with implementing these commands. Although it will be a non-bridge node which issues the ZIP takedown and ZIP bringup commands, it is envisioned that an application program will be provided which performs the following functions: issues the ZIP takedown command, waits for the ZIP bringback time, and issues a ZIP bringup command with the new zone name. Thus only the application program will have to implement these ZIP commands.

Note that ZIP provides no means of renaming an entire zone -- each network on the zone must be renamed individually.

## Listing Zone Names

ZIP provides ways for normal nodes to obtain information about the zone structure of the internet. For one, normal nodes are free to issue ZIP queries to bridges. Through ZIP queries, normal nodes can obtain the zone name associated with any network on the internet, including their own. Since ZIP queries and responses are delivered only on a "best effort" basis, the normal node will also have to implement a timeout-and-retry mechanism to guarantee a response. In addition, the ZIP query mechanism provides no simple way of obtaining a complete list of all the zone names on the internet. For these reasons ZIP provides additional commands through which normal nodes can obtain zone information.

Obtaining zone information is a typical request/response type transaction. The normal node requests some information from a bridge, and the bridge responds with that information. For this reason, ATP is used for these commands. However. to prevent it from being necessary to implement a full ATP responder in all bridges, the following restrictions apply to these requests: (1) they are of the "atleast once" type, (2) they are short enough to fit entirely in the ATP user bytes, and (3) they ask for only a single response packet.

Two ATP requests are provided by ZIP. They are both sent to the ZIS in any bridge on the local network. The ATP retry count and interval are left up to the particular implementation.

The *GetZoneList* command is used to obtain a list of all the zones on the internet. Since all the zone names may not fit in one ATP response packet, the request contains an index at which to start including names in the response (zone names in the bridge are assumed to be indexed from one on up). To obtain the complete zone list, a non-bridge node sends a series of ATP requests. The first of these requests specifies an index of one. The response user bytes contain the number of zone names in that response packet, and whether there are any more zones whose names didn't fit in the response. If there are, the node should send out another request, with the index equal to the index sent in the last request plus the number of names in the last response. In this way a node can obtain the complete zone list.

It is an important feature of this command that zone names do not cross response boundaries. It is also important that, if more than one request is necessary in order to

obtain the complete zone list, then all requests must be sent to the same bridge. This is because different bridges may have their zone lists stored in different orders. Note that a zero byte response will be returned by a bridge if the index specified in the request is greater than the index of the last zone in the list (and the user bytes will indicate "no more zones"). Although during periods when zone names are being changed, a station may receive a particular zone's name more than once, a bridge should make every attempt to send a particular zone's name only once in response to a GetZoneList command.

The *GetMyZone* command is used to obtain the name of the zone in which the requesting node resides (i.e. what '*' is equivalent to). This command is essentially a simplified GetZoneList request, where only one zone name is returned. A zero byte response will be returned if the bridge does not currently know the zone name of the zone (a temporary condition).

A bridge which is in the process of changing the zone name corresponding to a network to which it is directly connected, is not expected to respond correctly to a *GetMyZone* command from a node on that network. This restriction applies only to the period during which the zone name is being changed.

## Packet Formats

Figure VIII-1 summarizes the ZIP packet formats. A ZIP packet is always sent with a short DDP header (i.e. on the local network). Queries are always sent to the ZIS in a bridge (or broadcast); replies are always sent from the bridge to the socket the query came from. Takedowns and Bringups are broadcast to the ZIS. The DDP type field in these four packets is set to 6 to indicate Zone Information Protocol. All four contain a header made up of a command byte (1=query, 2=reply, 3=takedown, 4=bringup) and a network number count (n), which should be equal to one in takedowns and bringups. Queries then contain n network numbers for which zone names are desired. Replies contain n network-number/zone-name pairs, with zone names being preceded by a length byte. Takedowns contain no data part; Bringups contain an unused network number field, and then the new zone name string.

The GetZoneList request contains a command code of 8 indicating "GetZoneList" and the desired starting index, both in the ATP user bytes. The response contains, again in the user bytes, a flag which is non-zero if this response contains the last zones in the zone list, and the number of zones contained in the data part. The GetMyZone request contains a command code of 7 in the user bytes (all other user bytes should be zero). The response will generally indicate one zone name in the user bytes, and contain that zone name.

## NBP Routing in Bridges

As indicated in the NBP chapter, bridges contain an NBP process which is responsible for the conversion of BrRq requests to a zone-wide broadcast of LkUp requests. The process is a straightforward one. The bridge obtains, from its ZIT, a list of all the networks in the desired zone. It then uses DDP to send a directed-broadcast LkUp request to each of those networks. This directed broadcast is a normal DDP packet (generally with a long header), with a DDP destination node of $FF. If the bridge is directly connected to the desired network, the packet should be broadcast on that net (LAP destination also $FF, and a short DDP header can be used). If not, the packet is sent to the "next bridge" field in the routing table entry for that network, which will forward the packet appropriately.

One DDP directed broadcast is sent for each of the networks in the zone. The DDP data part is exactly the same as that from the BrRq, except the command field is changed to LkUp. Note that NBP is defined such that the bridge's NBP process does not participate in any way in the responding process (the response is sent directly to the requestor through DDP). Also note that in addition to BrRq requests, bridges will also receive normal LkUp's on the NIS. These should not be forwarded.

## Zone Name Assignment

RTMP defined the structure of a bridge's *port descriptor*. ZIP adds a zone name field to the port descriptor definition. Zone names are set into the port descriptors of bridge ports, and are then dynamically propagated out through ZIP to all bridges on the internet.

Only certain bridge ports need have zone names associated with them. These are the ports connected to AppleTalk networks. For each AppleTalk network, at least one bridge on that network must be configured with the network's zone name; all other bridges could have a zone name of NIL in their port descriptor associated with that net. If more than one bridge is configured with a non-NIL zone name for a given network, these names must be the same. Furthermore, only bridges which are seed bridges for purposes of specifying the network number should contain non-NIL zone names (i.e. if a bridge is not a seed bridge for routing information it should not be a seed bridge for zone information either).

Note that when the zone name of a network changes through a ZIP bringup command, all bridges on that net which have a non-NIL name in that net's port descriptor should be sure to change that name to the new one.

## Timer Values

There are two parameter values associated with ZIP which must be specified. The first of these is the value for the query retransmission timer. This is defined as equal to the Send-RTMP timer, or ten seconds.

The second parameter is the ZIP bringback time -- the minimum time required between bringing a network down and bringing it back up with a new zone name. Since this is a constant regardless of internet topology, the worst-case internet must be used in determining it. This value has been somewhat conservatively defined since changing a network's zone name will be a very rare event. The value for the ZIP bringback time is ten minutes.

## Implementation Notes

Although ZIP and RTMP are two separate and distinct protocols, changes in the ZIT are directly related to changes in RTMP's routing table. For this reason, bridge implementations may wish to combine the ZIT and routing table into one joint data structure, and the ZIP and RTMP processes into one joint module. This is perfectly acceptable.

ZIP query          ZIP reply          ZIP takedown          ZIP bringup

LAP Header

| ZIP query | ZIP reply | ZIP takedown | ZIP bringup |
|---|---|---|---|
| Destination node | Destination node | Dest node = $FF | Dest node = $FF |
| Source node | Source node | Source node | Source node |
| LAP type = 1 | LAP type = 1 | LAP type = 1 | LAP type = 1 |

DDP Header

| | | | |
|---|---|---|---|
| — Length — | — Length — | — Length — | — Length — |
| Dest. skt = 6 | Destination skt | Dest. skt = 6 | Dest. skt = 6 |
| Source socket | Source socket | Source socket | Source socket |
| DDP type = 6 | DDP type = 6 | DDP type = 6 | DDP type = 6 |

ZIP Header

| | | | |
|---|---|---|---|
| ZIP command = 1 | ZIP command = 2 | ZIP command = 3 | ZIP command = 4 |
| Network count | Network count | Ntwk. count = 1 | Ntwk. count = 1 |

ZIP Data

| ZIP query | ZIP reply | | ZIP bringup |
|---|---|---|---|
| — Network 1 — | — Network 1 — | | — 0 — |
| — Network 2 — | Zone 1 length | | Zone length |
| etc. | Zone name 1 | | Zone name |
| | — Network 2 — | | |
| | etc. | | |

ZIP GetZoneList
or GetMyZone

to ZIS (6) in ABridge,
DDP type = "ATP" (3)

GetZoneList or
GetMyZone Reply

to requestor,
DDP type = "ATP" (3)

| ATP Header | ZIP GetZoneList or GetMyZone | GetZoneList or GetMyZone Reply |
|---|---|---|
| | TReq=01000000 | TRsp=10010000 |
| | Bitmap=00000001 | Sequence no.=0 |
| | — TID — | — TID — |
| ATP User Bytes | GZL = 8, GMZ = 7 | LastFlag        (0 if GMZ) |
| | 0 | 0 |
| | — StartIndex — (0 if GMZ) | Number of — zones in packet |
| ATP Data | | Zone 1 length |
| | | Zone name 1 |
| | | etc. |

Figure VIII-1. ZIP packet formats

# IX. Printer Access Protocol (PAP)

## About Printer Access Protocol

The AppleTalk Printer Access Protocol (PAP) is a session level protocol intended for use between workstations and print servers. PAP is a connection-oriented protocol which handles connection setup, maintenance and teardown in addition to data transfer. Multiple connections are allowed at both the workstation and server ends.

PAP envisions a print server node as containing one or more processes, referred to in this document as servers, which are accessed by workstations through PAP. Each of these server (processes) makes itself visible over the network by opening a service listener socket (SL socket) on which the corresponding server registers its name(s).

A PAP client in a workstation issues a PAPOpen call which initiates a connection-establishment dialogue with a server. The client specifies the server by its complete name; in order to process the PAPOpen call, PAP itself calls NBP to obtain the address of the server's listener socket from the server's name. PAP allows implementations in which the workstation's PAP client performs the NBP lookup directly and then makes the PAPOpen call with the address of the server's service listener socket.

Once a connection has been opened to the server, the PAP client at either end of the connection can receive data from the other end by issuing PAPRead calls, and can write data to the other end through PAPWrite calls. PAP uses ATP transactions (in exactly-once mode) to transfer the data.

When the data transfer has been completed a PAPClose call is issued by the PAP client on either end to close the connection.

At any time, the PAP client in the workstation can issue a PAPStatus call to find out the status of a server. PAP does not restrict the syntactic or semantic structure of this status information beyond specifying that, in the PAP packets, it is a string of at most 255 bytes preceded by a length byte. Note that the PAPStatus call can be issued even before a connection has been set up to the server.

PAP is not a symmetric protocol. There are several PAP calls for use only in server nodes. The first of these is the SLInit call. This is issued by each server (in a server node) when it is first started up, after it has completed its initialization. The SLInit call opens a service listener socket (an ATP responding socket) in the server node, and causes the server's name to be registered on that socket through NBP. Multiple SLInit calls may be issued to the PAP in the same server node -- each SLInit call opens a new service listener socket and registers the server name provided in the call on that socket.

A second call is used by the PAP client in the server to indicate to the node's PAP connection arbitration code that this client is ready to accept a new connection through a particular service listener socket opened via a prior SLInit call. This GetNextJob call primes the connection arbitration code to accept another connection establishment request from a workstation.

An SLClose call can be issued by a server client to close a given service listener socket and shut down any active connections for that client.

Two calls, PAPRegName and PAPRemName, can be used by a PAP client in the server node to respectively, register and remove (deregister) a server name for a specified service listener socket. This might be necessary for giving the server (on a particular service listener socket) more than one name, or to change the server name (e.g. at server setup time) associated with a particular service listener socket.

One of the situations that PAP must deal with is the well-known case of half open connections. Such a connection is said to exist when one of the connection ends "dies" (or terminates the connection without informing the other end). Half open connections must be detected and torn down/closed. For this purpose, PAP maintains a connection timeout (at each end). Furthermore, each end of an open connection must send "tickling" packets to the other end on a periodic basis. The purpose of these packets is to inform the other end that the sender's end is open and "alive". The receipt of any packet on a connection resets the connection timer at the receiving end. If the connection timer expires (i.e., no packets have been received since the timer was last reset) then the decision is made that the other end is dead and the connection end is torn down.

A detailed discussion of the PAP protocol, its interaction with (i.e., use of) NBP and ATP, and the PAP client interface follows.


## The Protocol

The basic model of a PAP-based server is that it processes jobs from workstations, where at any given time it can be processing a specific maximum number of jobs (this number is server implementation dependent and is not a PAP parameter). While the server is processing this maximum number of jobs, requests for initiating further jobs are not accepted (the requesting workstation is informed that the server is busy). While a server is processing a job, a connection is said to be open between the workstation being served and the server. There is a one-to-one correspondence between open connections and jobs being processed by the server. When the server is done with a particular job, the corresponding connection is closed and the server may, at its discretion, notify its PAP that it is able to accept another request for service from any workstation.

When a server process is first started, it goes through its internal initialization and then issues an SLInit call to its PAP code. This causes PAP to call ATP to open an ATP responding socket. This is the service listener (SL) socket for that server. Then, PAP calls NBP to register the server's name(s) and bind them to the SL socket. An ATP Receive-a-Request call is then issued by PAP on this socket (so that the server can respond to PAPOpen or PAPStatus request packets). But the server itself (a PAP client in the server node) may still not be ready to accept a job/connection. PAP will still not accept connection opening requests through this SL socket. The server is said to be in a BLOCKED state. (see Figure IX-1).

After the SLInit call completes, the server process issues a series of GetNextJob calls to indicate that the server is ready to accept jobs. One such call is issued for each job it can accept at the time. The server is now in the WAITING state and is ready to accept jobs/connections.

PAP can support multiple servers within one node. Each of these servers is made available on a unique SL socket set up through a corresponding SLInit call.

PAP uses NBP to name and find a server's SL socket. Apart from these operations, all packets exchanged by PAP are sent through ATP, i.e. they are ATP request or response

packets. Each such PAP packet contains in the ATP user bytes a one-byte quantity that indicates its PAP-type. Thus, for instance reference will be made to an ATP transaction of PAP-type OpenConn, etc.

### Connection Establishment (Opening) Phase

A connection is a logical relationship between two PAP code entities (one in the workstation and the other in the server node). Data can be exchanged by two PAP clients only after a connection has been established/opened. Since PAP uses ATP to transfer data. the two communicating PAPs must, in the connection establishment phase, discover the address of the ATP responding socket of the other connection end. Also, the amount of data that can be transferred in an ATP transaction is of a maximum size equal to the available receive buffers at the end issuing the read requests. This maximum size (called the "flow quantum") is sent by each end to the other in the connection establishment phase.

Connection establishment is initiated by a PAP client in a workstation by issuing a PAPOpen call. Such a client provides as a call parameter the complete name of the server. The PAP code obtains the complete internet address of the server's SL socket by issuing an NBP Lookup call. It then opens an ATP responding socket $R_w$, generates an 8-bit connection identifier ConnID and then sends a transaction request (TReq), with PAP-type OpenConn, to the server's SL socket. This packet contains the ConnID, the address of socket $R_w$, the flow quantum for the workstation, and a wait time used by the server for arbitration (discussed later). All packets related to this connection (sent by either end) must contain this connection identifier. Packets with different connection ID's received through the sockets associated with the connection should be ignored by PAP. The workstation should be sure to generate the connection ID's in such a way as to minimize the likelihood of any two connections opened by that workstation having the same ID (especially connections established at about the same time).

When an ATP TReq of PAP-type OpenConn is received at the server's SL socket, PAP executes a connection acceptance algorithm (see figure IX-1). If the server is BLOCKED (i.e. there are no outstanding GetNextJob calls), then its PAP responds to the "OpenConn" with an ATP response of PAP-type OpenConnReply indicating "Server busy". Included in the OpenConnReply is a status string which is passed to the client and can be used for further detail on the busy state.

If however, the server is in the WAITING state (i.e. there are one or more pending GetNextJob calls), then upon receiving an OpenConn (the first one since the server went into the WAITING state), its PAP goes into an arbitration (ARB) state for a fixed amount of time (two seconds). In the ARB state, the PAP receives all incoming "OpenConns" and tries to find the ones corresponding to the workstations that have been waiting for a connection for the longest amount of time. The idea is to implement a fairness scheme that accepts the requests generated by these stations over those from more recent entrants to the contest.

The time, in seconds, for which a station has been waiting (called the WaitTime) is sent with the OpenConn. When the first OpenConn is received since the server went to the WAITING state, the WaitTime value from that request is loaded into a variable associated with one of the pending GetNextJob calls. This GetNextJob call is marked as being tagged. During the ARB interval, whenever a new OpenConn request is received. the server examines all the pending GetNextJob calls to see if any one of them is not tagged. If such a free pending GetNextJob call is found, then the WaitTime of the just received OpenConn is saved with that GetNextJob which is then marked as tagged. If no free

GetNextJob call is found among the pending calls, then PAP compares the just received OpenConn's WaitTime with the values saved in all the tagged pending GetNextJobs. If it is less than all of them, PAP responds to the just-received request with an OpenConnReply indicating "server busy." If it is greater than one (or more), PAP associates the new request with the GetNextJob with the smallest saved WaitTime, replacing that WaitTime with the one from the new request.

At the end of the ARB interval, PAP opens ATP responding sockets $R_s$ for each connection request still associated with a GetNextJob and sends ATP responses of PAP-type OpenConnReply indicating "connection accepted" to the selected (but still pending) requests. These carry the ConnID received in the "OpenConn", the address of socket $R_s$ and the flow quantum of the server end (the flow quantum value for the server end is set by the SLInit call issued when the server is initialized). The corresponding connections are now open, and the jobs from the corresponding workstations can now be processed.

At the end of the ARB state, if there are no pending GetNextJobs then the server goes to the BLOCKED state, else it goes to the UNBLOCKED state. In the BLOCKED state there are no pending GetNextJob calls and the server cannot accept incoming OpenConn requests. However, if it is in the UNBLOCKED state then there are pending GetNextJob calls and the server can still accept additional connections/jobs.

Note that if the server is in the UNBLOCKED state, it has just been through the ARB state and has already opened connections to all workstations that have been waiting for a connection. Thus in the UNBLOCKED state, upon receiving the next OpenConn request it is not necessary to go into the ARB state -- as long as it has pending GetNextJob calls, the server accepts all incoming OpenConn requests and sets up connections immediately. As soon as it runs out of pending GetNextJob calls, it goes to the BLOCKED state until a GetNextJob is issued -- at which time it must go into the WAITING state.

At the workstation end, if a response of PAP-type OpenConnReply is received indicating that the server is busy (i.e. in the BLOCKED state), then that end's PAP waits some time (around 2 seconds) and issues another connection opening transaction. Each time it repeats this process it updates a "wait time" -- the time in seconds that it has been trying to open the connection. The current value of this WaitTime is sent with each OpenConn. Each of these OpenConn ATP transaction requests is issued with a retry count of 5 and retry interval of 2 seconds. The workstation's PAP should provide some way for its client to abort a PAPOpen request, but should otherwise keep trying until the connection is opened.

Data Transfer Phase

Once a connection has been opened PAP's data transfer phase is initiated. In this phase, PAP has two functions: to actually transfer data over the connection, and to detect and tear down half-open connections.

The detection of half-open connections is done by maintaining a connection timer (of duration 2 minutes) at each end of a connection. This timer is started as soon as the connection is opened. Whenever a packet of any sort is received from the other end of the connection, the timer is reset. If the timer expires (clearly, without receiving a packet from the other end) the connection is torn down. The presumption is made that the other end has "died", has closed its connection or has become otherwise unreachable (for instance if an internet has become partitioned).

For this to work properly, it is important that even though no data is being exchanged on the connection, PAP exchanges control packets to signal that the connection ends are alive. This process is referred to as "tickling" and the control packets are called "tickling packets". For this purpose, as soon as a connection is established, each end starts an ATP transaction with PAP type Tickle. This transaction, known as the "Tickle" transaction, has a retry count of infinity and a retry time interval of half the connection timeout. The "Tickles" must be ATP transactions of at-least-once type. Tickle packets are sent to the other end's ATP responding socket (i.e. $R_s$ or $R_w$). The receiver of such a packet must reset its connection timer but must not send a transaction response. These "Tickle" transactions are cancelled by each end when the connection is closed.

The basic data transfer model used by PAP is "read-driven." When the PAP client at either end of the connection wishes to read data from the other end, it issues a PAPRead call. This call provides PAP with a read buffer (of size equal to this end's flow quantum) into which the data is to be read. As a consequence of this call, PAP calls ATP to send an ATP transaction request with PAP-type SendData, and the ATP bitmap reflecting the size of the call's read buffer. This transaction is issued with a retry count of "infinite" and a retry time interval of 15 seconds. It is sent to the other end's ATP responding socket. To prevent duplicate delivery of data to PAP's clients, all ATP transactions for the transfer of data use ATP's exactly-once mode and a sequence number. This is described in detail in the section entitled "Duplicate Filtration."

The receipt of an ATP TReq packet with PAP-type SendData implies that there is a pending PAPRead at the other end. This "send credit" can be remembered by the PAP code, and used to service any pending or future PAPWrite calls issued by its client.

When a PAP client (at either end) issues a PAPWrite call, PAP examines its internal data structures to see if it has received a "send credit". If it has, then it takes the data from the PAPWrite and sends it in ATP response packets with at most 512 bytes of ATP data each and of PAP-type Data (the EOM-bit is set in the last of these ATP response packets). If no send credit has been received, then PAP queues the PAPWrite call and awaits a "send credit" from the other end (i.e. the receipt of an ATP request of PAP-type SendData from the other end). The amount of data to be sent in a PAPWrite call cannot exceed the flow quantum of the other end (PAPWrite calls that violate this restriction return immediately with an error message).

When a PAP client issues the last PAPWrite call for a particular job, it should ask PAP to send an End-of-File (EOF) indication with that call's data. The EOF indication is delivered to the PAP client at the other end (as part of the received information for a PAPRead call); this indication notifies the client that the other end is through sending data on this connection. Note that for this purpose the client may issue a PAPWrite call with no data to be sent; in this case, just an EOF indication is conveyed to the client at the other end.

## Duplicate Filtration

As described in the ATP chapter, in the case of internets ATP exactly-once mode does not guarantee exactly-once delivery of requests -- it only guarantees that if a duplicate request is delivered to an ATP client it can be ignored because all responses to it have been successfully received by the other side. PAP uses a sequence number in read (SendData) requests to enable it to detect these duplicates and ignore them. Since PAP maintains only one outstanding read request at a time, this can be done in quite a simple way.

All read requests contain a sequence number in the last two ATP user bytes. The sequence number starts at 1 with the first read and takes on successive values up to 65535 before wrapping back around to 1 again. The value zero is reserved to mean "unsequenced." Any SendData request received with a sequence number of zero should be accepted by PAP without duplicate checking. This is for compatibility with previous versions of PAP. If the sequence number is non-zero, PAP should verify that the sequence number is equal to one more than the sequence number of the last SendData request received. If this is not the case, the packet should be ignored as a duplicate of a previous request. Each side of the PAP connection must independently maintain both a sequence number for its SendData requests and a sequence number for the last SendData request received from the other side.

## Connection Termination (Closing) Phase

When the PAP client at either end issues a PAPClose call, PAP closes the connection. Typically, after the workstation's PAP client has completed sending all data to the server and received an EOF in return, it will issue the PAPClose call. An ATP transaction request is sent to the other end with PAP-type "CloseConn". An end receiving a "CloseConn" should immediately send back, as a courtesy, an ATP transaction response of PAP-type "CloseConnReply". To close a connection's end, it is important to cancel any pending ATP transactions issued by that end, including the "tickle" transaction. An end receiving a "CloseConn" must cancel its pending ATP transactions for that connection as soon as it is able to do so.

At the server end, the receipt of the CloseConn will cause the connection to be torn down, but the server may continue to process the data pertaining to the job. When this is completed, the PAP client in the server may if it wishes to accept another job issue a GetNextJob call. Note that in fact the server may issue a GetNextJob call at any time in order to signal to its PAP code its willingness to accept another job. These GetNextJob calls are queued up by PAP and used to accept incoming OpenConn requests as discussed above in the Connection Establishment Phase.

A server may also close all open connections by issuing an SLClose call, which deregisters all names and closes its SL scoket.

## Status Gathering

Additionally, PAP supports status querying of the server through the PAPStatus call. It is not necessary to have a connection opened to the server to issue this call -- a workstation client can issue this call at any time. The call results in a SendStatus request packet being sent to the server specified in the call (the server can be specified by name, in which case PAP will call NBP to determine the server's address). The request is sent to the server's SL socket. The server's PAP responds with a StatusReply packet which contains a Pascal-format string (length byte first) specifying the server's status. This is done without passing the request to the PAP-client in the server. The PAP client in the server must have previously provided the status string to PAP through an SLInit or HeresStatus call. The HeresStatus call, which is implementation dependent, should be made by the PAP server client whenever the printer's status changes. Note that this status string is also returned by PAP in OpenConnReply packets.

## PAP Packet formats

PAP uses both NBP and ATP. The use of NBP is strictly for the purpose of registering a name on the server's SL socket and, given a server's name, for determining the address of its SL socket.

Packets sent by ATP in response to PAP calls include a PAP header. This is built using the user bytes of the ATP header, and in some cases by sending four or more bytes of PAP header in the data part of the ATP packet.

In all cases, the first of the ATP user bytes is the ConnID (except for SendStatus requests and Status replies for which this byte must be equal to zero). The second ATP user byte is the PAP-type of the packet. A list of PAP-type values follows in the next section. For packets of PAP-type equal to Data, the third ATP user byte is the EOF indication (non-zero indicates end-of-file). For packets of type SendData, the third and fourth bytes are the sequence number (high byte first). OpenConn and OpenConnReply packets contain, as part of the ATP data, the ATP responding socket numbers and the flow quantum to be used for the connection. The OpenConn request additionally contains the WaitTime; the OpenConn reply contains the open result and the status string. StatusReplies contain just the status string.

Figure IX-2 illustrates the PAP header for the different types of PAP packets.

## PAP Type Values

The permissible PAP Type field values are:

        OpenConn = 1
        OpenConnReply = 2
        SendData = 3
        Data = 4
        Tickle = 5
        CloseConn = 6
        CloseConnReply = 7
        SendStatus = 8
        StatusReply = 9

Values returned in the error code field of a OpenConnReply are:

        NoError = 0            { No error - connection opened }
        PrinterBusy = $FFFF    { Printer busy }

## The Client-PAP Interface

In this section we take each PAP call and list the parameters the client must pass and the significant interface-level aspects of each call.

### PAPOpen

This call is issued by a PAP client in a workstation to start/open a connection to a specified server.

• Call parameters:

> an indication as to the printer to which a connection should be opened (entity name or printer's SL socket address)
> flow quantum (number of 512 byte buffers for reads)
> a buffer in which the open status string is to be returned

• Returned parameters:

> result code
> connection refnum

The open status string should be returned by PAP at any time it is received in a OpenConnReply, not just upon call completion. The client must use the connection refnum in order to refer to this connection in subsequent calls.

## PAPClose

This call is issued by the PAP client in a workstation to close the connection specified by its connection refnum.

• Call parameters:

> connection refnum

• Returned parameters:

> result code

## PAPRead

This call is issued by the PAP client at either end to read data from the other end over the connection specified by the connection refnum.

• Call parameters:

> connection refnum
> buffer into which to read the reply

• Returned parameters:

> result code
> size of the data read
> the end-of-file indication

It is important to note that PAP assumes that the buffer into which the reply is to be read is equal to (no smaller than) the flow quantum specified in the PAPOpen call.

## PAPWrite

This call is issued by the PAP client at either end to write data to the other end over the connection specified by the connection refnum.

• Call parameters:

> connection refnum
> buffer with the data to be written
> size of the data to be written
> end-of-file indication

• Returned parameters:

> result code

If the data size is bigger than the flow quantum of the other end (value received during the connection establishment phase) the call will return with an error.

## PAPStatus

This call is used by a PAP client in the workstation to determine the current status of the print server. It can be used at any time regardless of whether a connection has been opened by the PAP client to the server or not. Upon completion this call returns a string with the status message sent by the print server.

• Call parameters:

> an indication as to the printer from which status is being requested (entity
> name or printer's SL socket address)
> a buffer in which the status string is to be returned

• Returned parameters:

> result code

In addition to these calls, the server uses the following four calls:

## SLInit

This call is issued by the PAP client in the server to open a service listening socket and to register the server's name on this service listening socket. The client can also provide an intial status string.

• Call parameters:

> the entity name of the printer
> the flow quantum for all connections to the printer (number of 512 byte buffers)
> a status string

• Returned parameters:

> result code
> the server refnum for the given printer

This server refnum must be used by the server when issuing subsequent GetNextJob calls in order to identify the listener socket for which the GetNextJob call is being made. Thus the PAP code in the server node must return a unique server refnum for each SLInit call.

GetNextJob

This call is issued by the PAP client in the server whenever it is ready to accept a new job through the listener socket specified by the server refnum.

• Call parameters:

       server refnum

• Returned parameters:

       result code
       a refnum to refer to this connection by in subsequent calls (read/write/close)

SLClose

This call is issued by the PAP client in the server whenever it wants to close down a given server process.

• Call parameters:

       server refnum

• Returned parameters:

       result code

PAPRegName

This call is used in server nodes only. It registers a name (as an entity name for the print server) on the server's listening socket corresponding to the specified server refnum.

• Call parameters:

       server refnum
       server name to register

• Returned parameters:

       result code

PAPRemName

This call is used in server nodes only. It deregisters a name from the server's listening socket corresponding to the specified server refnum.

• Call parameters:

       server refnum

r:

t:

```
                    ┌──────────┐
                    │  Start   │
                    └──────────┘
                         │
                   Server issues
                   SLInit  call
                         │
                         ▼
                    ┌──────────┐
                    │ Blocked  │
                    └──────────┘
     Server issues                    No pending
     GetNextJob                       GetNextJob
     call                             calls

  ┌──────────┐    End of Arbitrate period    ┌──────────┐
  │ Waiting  │      and no pending            │Unblocked │
  └──────────┘      GetNextJob  call          └──────────┘

     OpenConn                        End of Arbitrate period
     request arrives                 with pending
     through the SLSocket            GetNextJob
                                     calls
                    ┌──────────┐
                    │ Arbitrate│
                    └──────────┘
```

OpenConn requests received through the SL Socket are ignored if the state is BLOCKED

OpenConn requests received through the SLSocket are serviced immediately when in the
UNBLOCKED state, until no more GetNextJob calls are pending (then go to BLOCKED state)

Figure IX-1.  Server State Diagram

| | OpenConn (TReq) | OpenConnReply (TResp) | Data (TResp) | Status (TResp) |
|---|---|---|---|---|
| ConnID | ConnID ··········· | ConnID ··········· | ConnID | 0 |
| PAPType | = OpenConn | = OpenConnReply | = Data | = Status |
| 0 | 0 | 0 | EOF | 0 |
| | ATP Responding Skt Number | ATP Responding Skt Number ············ | data bytes | \<unused\> |
| | Flow quantum | Flow quantum | data bytes | \<unused\> |
| | Wait Time | Result | data bytes | \<unused\> |
| | | Status String | data bytes | Status String |

ATP User Bytes

ATP Data

8 bits

| | Tickle (TReq) | CloseConn (TReq) | CloseConnReply (TResp) | SendStatus (TReq) | SendData (TReq) |
|---|---|---|---|---|---|
| ConnID | ConnID | ConnID | ConnID | 0 | ConnID |
| PAPType | = Tickle | = CloseConn | = CloseConnReply | = SendStatus | = SendData |
| | 0 | 0 | 0 | 0 | Sequence Number (two bytes) |
| | 0 | 0 | 0 | 0 | |

ATP User Bytes

Figure IX-2. PAP packet format

# X. Echo Protocol (EP)

**About Echo Protocol**

The Echo protocol is implemented on a particular node as a statically-assigned socket (socket number 4 known as the Echoer socket) and an Echoer process on this socket. The Echoer listens for packets received through this socket. Whenever a packet is received (see Figure X-1 for the packet format) the echoer examines its DDP protocol type and the size of the DDP data in this packet. If this protocol type field is not equal to 4 (the DDP protocol type for the Echo protocol) or the DDP data length is zero then the Echoer discards the packet and ignores it. However, if the just received packet has a DDP protocol type equal to 4 and its DDP data length is one or more, then the Echoer examines the first byte of the DDP data (this is the Echo Protocol header). If this is equal to 1 (Echoer Function = Echo request) then the Echoer changes this field to 2 (Echoer Function = Echo reply) and calls DDP to send the packet back to the original sender.

A client process wishing to use the Echo protocol's service must first determine the internet address of the node it wishes to obtain the echo from (it will probably use NBP for this purpose). Then it calls DDP to send an Echo request packet to the Echoer socket in that node. The client can send the datagram through any socket it has opened (the Echo response will come back through this socket). The client then waits for the receipt of the Echo reply packet. Note that the client can set the Echo data part of the request packet to any pattern it desires and then examine the data in the reply (which will be the data sent in the request). This may be used by the client, for instance, to distinguish among the replies to various Echo requests it may have sent.

There are several situations in which an Echo reply is never received by the client. The Echo protocol packets could get lost in the network system. The target node may not have an Echoer. The target node may be unreachable or down. The client must use prudence in determining how long to wait for the echo reply before concluding that one of these situations exists. It might choose to try several times before concluding that the remote node will not respond at all.

This simple protocol can be used for two important purposes. In the first place, it can be used by any DDP client to determine if a particular node, known to have an Echoer, is accessible over an internet. A more significant use is to obtain an estimate of the round trip time for a typical packet to a remote node, usually a server. This is very useful in developing client-dependent heuristics for estimating the timeouts to be specified by clients of ATP, ASP and other higher level protocols. This is in general a difficult problem in the case of internets.
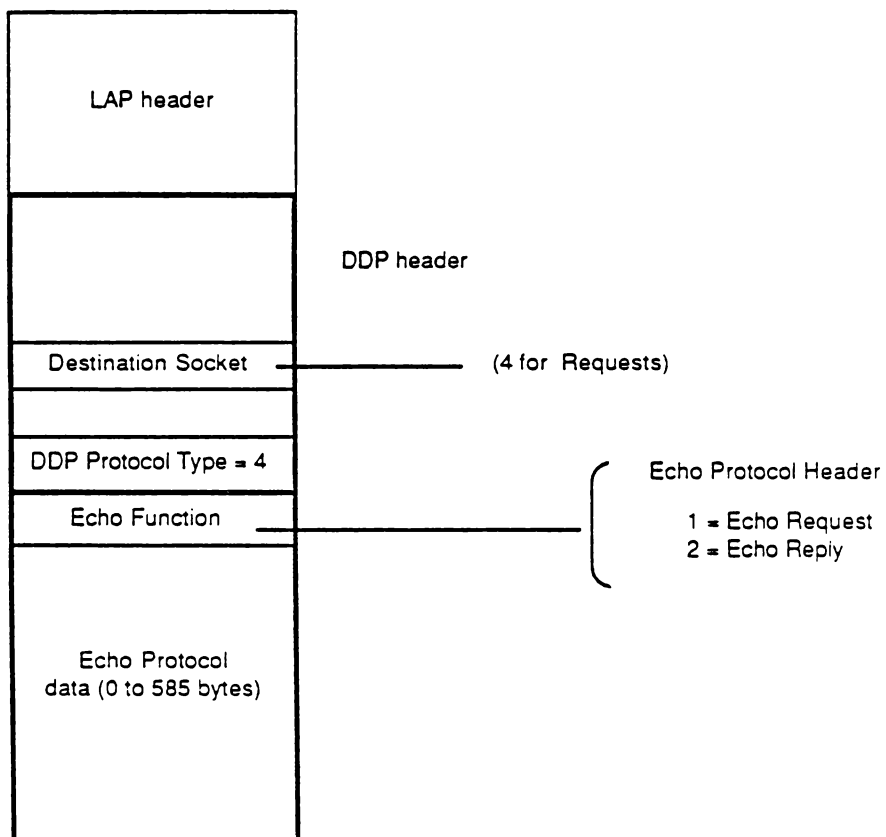
Figure X-1. Echo Protocol packet

# XI. AppleTalk Session Protocol (ASP)

This chapter specifies version 1.0 of the AppleTalk Session Protocol. It supersedes the preliminary draft document version 1.0/3. The protocol as specified herein is the final version 1.0 of this protocol. This protocol was developed in joint work by Apple Computer and Centram Systems West.

## About AppleTalk Session Protocol

A general conceptual model encompassing a wide variety of higher-level network services consists of a workstation issuing a sequence of commands to a server which executes these commands and returns the corresponding results to the workstation. An important example of such a service is a filing service through which file system commands can be conveyed to a file server for their execution.

At the transport level, the AppleTalk architecture provides a reliable transaction service via the AppleTalk Transaction Protocol (ATP) that can be used by such higher-level services for the conveyance of commands to servers. Nevertheless, ATP does not provide the full range of transport services needed by such higher-level protocols. In this document we describe the *AppleTalk Session Protocol* (ASP) designed specifically for the use of these higher-level protocols.

ASP is a client of ATP; it adds value to ATP to provide the level of transport service needed for higher-level workstation-to-server interaction.

## Sessions, Commands and Command Replies

Central to ASP is the concept of a *session*. Two network entities, one in a workstation and the other in a server can set up an ASP session between themselves. This is a logical relationship identified by a unique *session identifier*. For the duration of the session, the workstation entity can, through ASP, send a sequence of *commands* on the session to the server entity. ASP ensures that the commands are delivered in the same order as they were sent and conveys the results of these commands (known as a *command reply* or simply a *reply*) back to the workstation entity.

It is important to note that ASP sessions are inherently asymmetrical. The process of setting up a session is always initiated by the workstation entity (when it wishes to use the server entity's advertised service). Once the session is set up, the workstation client of ASP sends commands on the session and the server client of ASP replies to the commands. ASP does not allow the server client of ASP to send commands to the workstation client. It does, however, provide an "attention" mechanism by which the server can inform the workstation that it is in need of attention.

Although ASP guarantees that commands issued by the workstation end of a session are delivered to its server end in the same order as they were issued, it is beyond the scope of ASP to ensure that the commands are in fact executed and completed in that order by the server end. This is clearly the responsibility of the ASP client at the server end.

Many workstation entities can have sessions to the same server entity at the same time. ASP uses the session identifier to distinguish between commands received on these various

sessions. The session identifier of a session is unique among all the sessions to the same server.

## What ASP Does Not Do

ASP does not in any way understand the syntax or the semantics of the commands sent by its workstation clients on a session. Commands are conveyed basically as blocks of bytes whose interpretation is left to the server end/client of the session. Likewise, the command replies are sent back over the session to the workstation client without any syntactic/semantic understanding/interpretation by ASP.

It has been an important goal in the design of ASP to make its client interface independent of the lower-level transport protocols. This makes the higher-level clients of ASP transportable from one network to another with a minimum of modification.

As an important consequence it is necessary to separate out from ASP both the mechanism by which a server entity advertises its service and the manner in which workstation entities look for this advertised service. How this is realized depends intimately on the transport and naming mechanisms of a particular network; ASP assumes that these functions are taken care of by the ASP clients themselves and are outside its scope.

For instance, let us discuss the mechanism used on AppleTalk. A server entity wishing to advertise its service on the AppleTalk network calls the AppleTalk Transaction Protocol (ATP) to open an ATP responding socket known as the *Session Listening Socket* (SLS). The server entity then calls the Name Binding Protocol (NBP) to register a name on this socket. [At this point, the server entity calls ASP, via the *SPInit* call discussed later, to pass it the address of the SLS. Then, ASP starts listening on the SLS for session opening requests coming in over AppleTalk.] Now, a workstation entity wishing to access this advertised service uses NBP to discover the address of the SLS (this is known as the *Entity Identifier*). Then, it calls the workstation ASP to open a session by sending a session opening packet to the SLS. [Note that the operations of setting up an SLS and of looking for the SLS via NBP are done outside the scope of ASP. ASP's participation starts with the process of setting up a session.]

ASP does not provide a user authentication mechanism. This is expected to be the responsibility of the higher-level protocol being used by the clients of ASP.

ASP does not provide a mechanism to allow the use of a particular session by more than one server entity. Such multiplexing of a session can be done by the ASP clients if the command identifiers are broken into ranges by the higher-level protocols and managed completely outside the scope of ASP. We recommend the use of separate sessions to access different services on the same server.

## ASP Services and Features

ASP provides the following services to its clients:

- setting up (*opening*) and tearing down (*closing*) sessions;
- sending *commands* (on an open session) to the server and returning its
  *command replies* (which might include a block of data);
- *writing* blocks of data from the workstation to the server end of the session;
- sending *attentions* from the server to the workstation.

In addition, the workstation client can ask ASP to obtain *service status information* from the SLS without opening a session. The status information (provided by the server entity) is not semantically intelligible to ASP.

## Opening and Closing Sessions

Before any sessions are opened, both workstation and server ASP clients should interrogate ASP to discover the maximum sizes of command and replies allowed by the underlying transport mechanism. This information may be used by both ends to determine whether or not the underlying transport services are adequate to their needs, or just to optimize their commands and replies.

After discovering the network address of the desired service's SLS (the entity identifier), the workstation client calls ASP to open a session to this service (the client passes the entity identifier in this call). As a result of this call, ASP sends a special *OpenSession* packet (an ATP request) to the SLS; this packet carries the address of a workstation socket to which session maintenance packets (discussed later) are to be sent. This socket will be referred to as the *workstation session socket* (WSS). If the server entity is unable to set up a session, it returns an error (in the ATP response packet known as an *OpenSessReply*), otherwise it returns a session acceptance indication together with a *session identifier*, and the number of the server-end socket of the session (this socket will be referred to as the *Server Session Socket* or SSS). In all further communication over this session all packets must carry this *SessionID* and must be sent to the SSS.

ASP allows protocol version number verification in this session opening dialogue. The ASP in the workstation sends an ASP *protocol version number* in the *OpenSession* packet. This identifies the version of the ASP protocol that the workstation is using. If the server's ASP is unable to handle this version it will return an error in the *OpenSessReply* and the session will not be opened.

A session can be closed by the ASP client at either end. This is done by issuing the appropriate command to that end's ASP which duly informs the other end and then immediately tears down the session. If the session termination was initiated by the workstation client, then a session termination notification is sent to the SSS (this is an ATP request which carries the *SessionID*). If it was initiated by the server client, then the notification is sent to the WSS, again as an ATP request carrying the *SessionID*.

Whenever a session is terminated, the ASP clients at both ends must be notified of this occurence so that appropriate higher-level actions may be taken. This is easily done at the server end since it is generally listening for incoming commands on the session. But at the workstation end (if the server end closed the session) this is done the next time that the ASP client tries to issue a command on that session. The actions taken by an ASP client, upon being informed of the closing of a session, could be of various sorts depending on the higher-level function. For instance, the server end might choose to free up resources allocated for that session. If the higher-level service is a filing service, it might decide to flush all files, etc., opened on that session.

## Session Maintenance

A session will remain open until it is explicitly terminated by the ASP client at either end or until one of the session's ends "dies" or becomes unreachable. ASP includes a mechanism known as *session tickling* which is initiated as soon as a session is opened. The idea is that

each end of the session periodically sends a packet (known as a *tickle* packet -- an ATP request) to the WSS or the SLS to inform the other end that it is alive and well. If either end fails to receive *any* packets on a session for a certain predefined *session maintenance time-out* it assumes that the other end has "died" or has otherwise become unreachable. At that time the session is closed.

A simple way for an ASP session end to generate tickle packets is to issue an ATP request with its retry count set to infinite. This ATP request must be cancelled when the session is closed.

## Session Requests on an Open Session

Once a session has been opened, the workstation client of ASP can send a sequence of commands over the session to the server end. These commands are delivered in the same order as they were issued at the workstation end, and replies to the commands are returned to the workstation end by ASP. These commands can be classified into two types which differ in the direction of the primary flow of data:

1)    ASP Commands: These are very similar in nature to ATP requests. The ASP workstation client sends a command (encoded in a variable size command block) to the server-end client requesting it to perform a particular function and send back a variable size command reply. Examples of such commands could vary from requests to open a particular file on a file server, to reading a certain range of bytes from an already opened file. In the first case a small amount of reply data is returned, in the second case a multi-packet reply might be generated. Each ASP Command translates into an ATP request sent to the SSS and the command reply is received as a possibly multi-packet ATP response. (see Figure XI-3) In any case ASP does not interpret the command block or in any way participate in the command's function. It simply conveys the command block, encoded in a higher-level format unknown to ASP, to the server end of the session, and returns the command reply to the workstation-end client. The command reply consists of a four-byte *command result* and a variable size *command reply block*.

2)    ASP Writes: In this case the ASP client in the workstation wishes to convey a variable size block of data (known as the *WriteData*) to the server end of a session and expects a reply. ASP uses ATP as its underlying transport protocol. Since ATP is a protocol in which a requesting end essentially "reads" a multi-packet block of data from the responding end, it is necessary to translate the ASP Write into possibly two transactions. First, ASP sends an ATP request to the SSS carrying the ASP Write's control information (this can be called the *Write command block*). [This is illustrated in Figure XI-5.] The server end examines this information and determines whether it wants to go ahead with "pulling" the data from the workstation end. If not, it returns an error in the ATP response packet (this error is conveyed to the workstation client as the four-byte *command result*). If it wants to pull the data, then the server end sends an ATP request to the WSS to "pull" the data from the workstation end. This latter request could generate a multi-packet ATP response carrying the write data to the server. The server end upon receiving the write data and performing the particular function requested in the ASP Write, then responds to the original ATP request with the appropriate error message (this error is conveyed to the workstation client as the four-byte *command result*).

Additionally, once a session has been opened, the server client can send an *attention request* to the workstation client. This is an ATP-ALO request sent to the WSS. The sole purpose of this request is to alert the workstation client as to the server's need for attention. ASP delivers two bytes of *attention data* (from the request's ATP user bytes) to the

workstation client and acknowledges the attention request (with an ATP response), but it is the workstation client's responsibility to act on the request. An example of the use of the attention mechanism might be if a server desires to notify a workstation as to a change in its status. Upon receiving the attention request, the workstation could then issue an ASP command to the server to find out the details of the status change.

## Sequencing and Duplicate Filtration on Sessions

ASP ensures the delivery on a session of commands and writes to the server's end in the same order in which they were issued at the workstation end. This is done by including a *sequence number* in the appropriate packets exchanged by ASP.

The use of sequence numbers also allows ASP to add robustness to the ATP-XO service. Although ATP-XO will guarantee that a request is delivered to the ATP client exactly once if the source and destination nodes are on the same AppleTalk, this might not be the case over an AppleTalk internet. In the latter case it is possible for a copy of the ATP Request that was delayed in a bridge node to be delivered as a duplicate after the original transaction has been completed. Thus a duplicate transaction will be delivered. This problem is inherent to transaction protocols and can be eliminated when the concept of a session, as with ASP, is introduced and the transactions belonging to a particular session carry a sequence number which can be used to filter out delayed duplicates.

## Getting Service Status Information

ASP provides an out-of-band service to allow its workstation clients to obtain a block of service status information from the SLS without the need for opening a session. In the server the status block is provided to ASP by the corresponding ASP client and is returned in reponse to *ASPGetServerStatus* commands received at the SLS.

## ASP Client Interface

Having provided the preceding overall description of ASP and its services, we can now proceed to a more detailed specification, first of the service interface offered by ASP to its clients at the workstation and server ends, and then of the internal packet formats and details of how ASP uses ATP.

ASP's service interface has been designed to make it as independent as possible of the underlying AppleTalk transport mechanisms. The primary motivation for this is to allow easy porting of the higher-level (ASP clients) protocols to other networks than AppleTalk and to simplify some of the problems in the design of internet gateways. Nevertheless, the internals of ASP are intimately related to the AppleTalk Transaction Protocol, and thus ASP itself is not directly portable to other networks.

## Server Side

To start with, the server's ASP client should issue an *SPGetParms* call to retrieve the maximum values of command block size and Quantum size. The *MaxCmdSize* is the maximum size command that can be sent to the server. The *QuantumSize* returned by this call is the maximum size reply that can be sent to a command or the maximum size of data that can be transferred in a *Write* request. Since ASP is built on top of ATP the value of

*MaxCmdSize* returned will be 578 bytes and *QuantumSize* will be 4624 bytes (8 ATP response packets with 578 data bytes each). For client-compatible session protocols implemented on other networks, these values will be different.

<u>*SPGetParms*</u>

>  <u>Inputs</u>:
>
>>  none.
>
>  <u>Outputs</u>:
>
>>  *MaxCmdSize*-- maximum size of a command block;
>>  *QuantumSize* -- maximum data size for a command reply or a write.
>
>  <u>Errors</u>:
>
>>  none.

Before any workstation can open a session with a server, the server's SLS must be established. This must be done by the ASP client at the server end. This client must issue an *OPEN-ATP-Responding-Socket* call to create the SLS. Then it must call NBP to register the appropriate server entity name on this socket. At this point, the SLS is set up so that workstations can discover its network address through an NBP lookup. However, workstations still can not open sessions with this service entity. For this purpose, the ASP client in the server must now issue an *SPInit* call to ASP (passing to ASP the network address, known as the *SLSEntityIdentifier*), followed by one or more *SPGetSession* calls.

<u>*SPInit*</u>

>  <u>Inputs</u>:
>
>>  *SLSEntityIdentifier* -- SLS network identifier;
>>  *ServiceStatusBlock* -- block with status information;
>>  *ServiceStatusBlockSize*-- size of status information block.
>
>  <u>Outputs</u>:
>
>>  *SPError* -- error code returned by ASP;
>>  *SLSRefNum* -- reference number for the SLS.
>
>  <u>Errors</u>:
>
>>  *TooManyClients* -- ASP implementation cannot support another client;
>>  *SizeErr* -- *ServiceStatusBlockSize* is greater than *QuantumSize*.

This call is issued by the ASP client after having opened and named the SLS. The call passes the (network dependent) *SLSEntityIdentifier* to ASP as well as a *ServiceStatusBlock*. This block is used to hold the service status information to be returned in reply to *GetStatus* requests received at the SLS. The *SLSEntityIdentifier* is the complete internet address of the SLS.

*SPInit* returns the *SLSRefNum* (this is unique among all SLS's on the same server node) which is used in the *SPGetSession* call to make reference to the SLS passed in the *SPInit* call.

## SPGetSession

### Inputs:

*SLSRefNum* -- reference number for the SLS.

### Outputs:

*SPError* -- error code returned by ASP;
*SessRefNum* -- session reference number.

### Errors:

*ParamErr* -- unknown *SLSRefNum*;
*NoMoreSessions* -- implementation cannot support another session.

This call is issued by the ASP client to allow it to accept an *OpenSession* command received on the SLS identified by the *SLSRefNum*. Each *SPGetSession* request authorizes ASP to accept one more *OpenSession* request. The call completes when such a request is received on the SLS and a corresponding session has been opened. The *SessRefNum* is returned to the server ASP client and must be used in all further calls to ASP that refer to this session. Clearly the *SessRefNum* must be unique among all sessions open to the server.

## SPCloseSession

### Inputs:

*SessRefNum* -- session reference number.

### Outputs:

*SPError* -- error code returned by ASP.

### Errors:

*ParamErr* -- unknown *SessRefNum*.

This call is issued by the ASP client to close the session identified by *SessRefNum*. As a result of this call that value of the *SessRefNum* is invalidated and can not be used in any further calls. Furthermore, all pending activity on the session is immediately cancelled. ASP clients who want all pending activity to be completed before the session is closed, should make sure of this before making the *SPCloseSession* call.

## SPGetRequest

### Inputs:

*SessRefNum* -- session reference number;
*ReqBuff* -- buffer for receiving the command block;
*ReqBuffSize* -- buffer size.

### Outputs:

*SPError* -- error code returned by ASP;
*ReqRefNum* -- request identifier;
*SPReqType* -- SP level request type;
*ActRcvdReqLen* -- actual size of the received request.

### Errors:

*ParamErr* -- unknown *SessRefNum*;
*BufTooSmall* -- *ReqBuff* is too small to hold the entire command block;
*SessClosed* -- session has been closed.

After a session has been opened, the ASP client in the server must issue *SPGetRequest* calls to provide buffer space for the receipt of requests on that session. The size of the buffer for receiving the command block sent with the request depends on the higher-level protocol, but need be no greater than *QuantumSize*.

After a request has been received, this call completes and returns a unique request identifier *ReqRefNum* and a one-byte quantity that identifies the SP level type of request. The permissible values of *SPReqType* are *Command*, *Write* and *CloseSession*. If the received command block does not fit in the *ReqBuff*, then ASP returns as much as will fit along with a *BufTooSmall* error.

When this call completes, the caller is given the size of the received command block in the parameter *ActRcvdReqLen*.

If the session times out and an *SPGetRequest* is pending, the call will complete with an *SPError* value of *SessClosed*. If no call is pending, the next *SPGetRequest* call issued on the session will complete immediately with an error.

## SPCmdReply

### Inputs:

*SessRefNum* -- session reference number;
*ReqRefNum* -- request identifier;
*CmdResult* -- four byte command result;
*CmdReplyData* -- command reply data block;
*CmdReplyDataSize* -- size of command reply data block.

### Outputs:

*SPError* -- error code returned by ASP.

Errors:

> *ParamErr* -- unknown *SessRefNum* or *ReqRefNum*;
> > bad *CmdReplyDataSize* (negative value);
>
> *SizeErr* -- *CmdReplyDataSize* is greater than *QuantumSize*;
> *SessClosed* -- session has been closed.

If the request returned by the *SPGetRequest* call has *SPReqType* = *Command* then the ASP client must respond to it with an *SPCmdReply* call to ASP. The value of *ReqRefNum* passed with this call must be exactly the same as that returned by the corresponding *SPGetRequest* call. Two items must be conveyed to the workstation end of the session: a four-byte *command result* and a variable size *command reply block*. The actual values, format and meaning of the *CmdResult* and of the *CmdReplyData* are transparent to ASP. Note that the *CmdReplyData* can not be of size greater than the *QuantumSize* returned by the *SPGetParms* call (i.e. *CmdReplyDataSize* must be no greater than *QuantumSize*), or else a *SizeErr* will be returned and no *CmdReplyData* will be sent to the workstation.

## SPWrtContinue

Inputs:

> *SessRefNum* -- session reference number;
> *ReqRefNum* -- request identifier;
> *Buffer* -- buffer for receiving the data to be written;
> *BufferSize* -- size of the buffer.

Outputs:

> *SPError* -- error code returned by ASP;
> *ActLenRcvd* -- actual amount of data received into *Buffer*.

Errors:

> *ParamErr* -- unknown *SessRefNum* or *ReqRefNum*;
> > bad *BufferSize* (negative value);
>
> *SessClosed* -- session has been closed.

If the request returned by the *SPGetRequest* call has *SPReqType* = *Write* then the ASP client must respond to it with either an *SPWrtContinue* or an *SPWrtReply* call to ASP. The value of *ReqRefNum* passed with these calls must be exactly the same as that returned by the corresponding *SPGetRequest* call.

Exactly how the ASP client decides which of these calls to make depends on the higher-level protocol, but here is a description of the general idea. Upon receiving a request of *SPReqType* = *Write*, the ASP client examines the command block received with the request. This should contain, in the format appropriate to the higher-level protocol, a description of the type and parameters of the higher-level write operation being requested. The ASP client should use this command block information to decide if it can successfully carry out the operation being requested. If the operation cannot be carried out it should issue the *SPWrtReply* call with the appropriate higher-level protocol value in *CmdResult* indicating its inability to execute the operation and the reason why. If however, the operation can be carried out, then the ASP client in the server should initiate the process of actually transferring the data to be written from the workstation end of the session. This is

done by issuing the *SPWrtContinue* call. A *SPWrtReply* call should also be issued upon completion of the write.

For instance, the higher-level client could be a filing protocol requesting the ASP client in the server to write a certain number of bytes to a particular file. If no such file exists, the server end should send back a "no such file" indication by issuing an *SPWrtReply* call. Otherwise it issues an *SPWrtContinue* call with a buffer into which the write data can be brought from the workstation, followed by an *SPWrtReply* once it has finished the write request.

The maximum size of the write data that will be transferred is equal to the *QuantumSize*. Providing a larger buffer is of no use.

*SPWrtReply*

    Inputs:

        *SessRefNum* -- session reference number;
        *ReqRefNum* -- request identifier;
        *CmdResult* -- four byte command result;
        *CmdReplyData* -- command reply data block;
        *CmdReplyDataSize* -- size of command reply data block.

    Outputs:

        *SPError* -- error code returned by ASP.

    Errors:

        *ParamErr* -- unknown *SessRefNum* or *ReqRefNum*;
                bad *CmdReplyDataSize* (negative value);
        *SizeErr* -- *CmdReplyDataSize* is greater than *QuantumSize*;
        *SessClosed* -- session has been closed.

This call is issued by the ASP client in the server in order to terminate either successfully or unsuccessfully a *Write* command received through *SPGetRequest*. With this call the ASP client provides ASP with the four-byte *CmdResult* and the variable size command reply data block *CmdReplyData* (at most *QuantumSize* bytes) to be conveyed to the workstation end client. If a *SizeErr* is returned, no *CmdReplyData* will be sent to the workstation.

*SPNewStatus*

    Inputs:

        *SLSRefnum* -- reference number for the SLS;
        *ServiceStatusBlock* -- block with status information;
        *ServiceStatusBlockSize* -- size of status information block.

    Outputs:

        *SPError* -- error code returned by ASP.

Errors:

> *ParamErr* -- unknown *SLSRefnum*
> *SizeErr* -- *ServiceStatusBlockSize* is greater than *QuantumSize*.

This call is used by the ASP client to update the *ServiceStatusBlock* first supplied in the *SPInit* call. The previous status information is lost. All subsequent *SPGetStatus* calls issued by workstations will retrieve the new status block.

## *SPAttention*

Inputs:

> *SessRefnum* -- session reference number;
> *AttentionCode* -- two-byte attention code (must be non-zero);

Outputs:

> *SPError* -- error code returned by ASP.

Errors:

> *ParamErr* -- unknown *SessRefnum* , zero *AttentionCode*
> *NoAck* -- no acknowledgment from workstation side

This call sends the attention bytes to the workstation and waits for an acknowledgment.

## Workstation Side

## *SPGetParms*

Inputs:

> none.

Outputs:

> *MaxCmdSize*-- maximum size of a command block;
> *QuantumSize* -- maximum data size for a command reply or a write.

Errors:

> none.

This call is exactly the same as the *SPGetParms* call for the Server side.

## *SPGetStatus*

Inputs:

> *SLSEntityIdentifier* -- SLS network identifier;

*StatusBuffer* -- buffer for receiving the status information;
*StatusBufferSize* -- size of this buffer.

Outputs:

*SPError* -- error code returned by ASP;
*ActRcvdStatusLen* -- size of status information received.

Errors:

*NoServer* -- server not responding;
*BufTooSmall* -- *StatusBuffer* is not big enough to hold entire status.

This call is used by a workstation ASP client to obtain status information corresponding to a particular SLS. If the status information received is too large to fit into the *StatusBuffer* provided with the call, then an appropriate *BufTooSmall* value of *SPError* is returned but as much of the status information as fits is put in the *StatusBuffer*.


## *SPOpenSession*

Inputs:

*SLSEntityIdentifier* -- SLS network identifier;
*AttnRoutine* -- Attention routine indicator (implementation dependent).


Outputs:

*SPError* -- error code returned by ASP;
*SessRefNum* -- session reference number.

Errors:

*NoServer* -- server not responding;
*ServerBusy* -- server cannot open another session;
*BadVersNum* -- server cannot support the offered version number.
*NoMoreSessions*-- no more sessions available at workstation

This call is issued by an ASP client after it has obtained the network address/identifier of the SLS via an NBP lookup. If a session is successfully opened, then a *SessRefNum* is returned to the caller and should be used on all subsequent calls referring to this session. If a session cannot be opened, for whatever reason, an appropriate *SPError* value is returned. *AttnRoutine* specifies, in an implementation-dependent manner, a routine to invoke upon receipt of an attention request from the server.


## *SPCloseSession*

Inputs:

*SessRefNum* -- session reference number.

Outputs:

*SPError* -- error code returned by ASP.

<u>Errors</u>:

> *ParamErr* -- unknown *SessRefNum*.

This call can be issued at any time by the ASP client to close a session previously opened via an *SPOpenSession* call. As a result of this call that value of the *SessRefNum* is invalidated and can not be used in any further calls. Furthermore, all pending activity on the session is immediately cancelled. ASP clients who want all pending activity to be completed before the session is closed, should make sure of this before making the *SPCloseSession* call.

### *SPCommand*

<u>Inputs</u>:

> *SessRefNum* -- session reference number;
> *CmdBlock* -- command block to be sent;
> *CmdBlockSize* -- size of command block;
> *ReplyBuffer* -- buffer for receiving the command reply data;
> *ReplyBufferSize* -- size of this buffer.

<u>Outputs</u>:

> *SPError* -- error code returned by ASP;
> *CmdResult* -- four byte command result;
> *ActRcvdReplyLen* -- actual length of command reply data received.

<u>Errors</u>:

> *ParamErr* -- unknown *SessRefNum*;
> *SizeErr* -- *CmdBlockSize* is larger than *MaxCmdSize*;
> *SessClosed* -- the session has been closed.
> *BuffTooSmall*-- the reply buffer cannot hold the whole reply

Once a session has been opened, the ASP client can send a command to the server end by issuing an *SPCommand* call to ASP. A command block of maximum size *MaxCmdSize* can be sent with the command. (This maximum command block size for ASP is 578 bytes). Thus, if *CmdBlockSize* is larger than this maximum allowable size, then the call completes with *SPError* equal to *SizeErr*; in this case, no effort is made to send anything out over the network to the server end.

In response to a command, the server end returns two quantities: a four-byte *CmdResult* and a variable length command reply which will be returned in the *ReplyBuffer*. The size of the command reply actually received is returned in *ActRcvdReplyLen*. Clearly, this can be no larger than *QuantumSize*, so it is possible that only part of the reply is returned in this call. It is the responsibility of the ASP workstation-end client to generate another command to retrieve the rest of the reply.

<u>SPWrite</u>

    <u>Inputs</u>:

        *SessRefNum* -- session reference number;
        *CmdBlock* -- command block to be sent;
        *CmdBlockSize* -- size of command block;
        *WriteData* -- data block to be written;
        *WriteDataSize* -- size of data block to be written;
        *ReplyBuffer* -- buffer for receiving the command reply data;
        *ReplyBufferSize*-- size of this buffer.

    <u>Outputs</u>:

        *SPError* -- error code returned by ASP;
        *CmdResult* -- four byte command result;
        *ActLenWritten* -- actual number of bytes of data written;
        *ActRcvdReplyLen* -- actual length of command reply data received.

    <u>Errors</u>:

        *ParamErr* -- unknown *SessRefNum*;
        *SizeErr* -- *CmdBlockSize* is larger than *MaxCmdSize*;
        *SessClosed* -- the session has been closed.
        *BuffTooSmall*-- the reply buffer cannot hold the whole reply

This call is made by the ASP client to write a block of data to the server end of the session. The call first delivers the *CmdBlock* (no larger than *MaxCmdSize*) to the server end client of ASP and as described above, then the server end can actually transfer the *WriteData* over or return an error (delivered in the *CmdResult*).

The actual amount of data sent over will be less than or equal to *WriteDataSize* and will in any case never be larger than *QuantumSize*. The amount of write data actually transferred is returned in *ActLenWritten*.

In response to a write, the server end returns two quantities: a four-byte *CmdResult* and a variable length command reply which will be returned in the *ReplyBuffer*. The size of the command reply actually received is returned in *ActRcvdReplyLen*. Note that this can be no larger than *QuantumSize*.

## Packet Formats and ASP Internals

In this section we spell out the internal details of ASP including packet formats. This is done by discussing the major operations and for each case presenting the internal packet exchanges and their structure.

<u>Opening a Session</u>

When the workstation client issues an *SPOpenSession* call, ASP issues an ATP-XO transaction request addressed to the SLS. .(See Figure XI-1) This ATP transaction request

packet is known as an ASP *OpenSess* packet. The server's ASP returns an ATP transaction response packet known as an *OpenSessReply* packet.

The *OpenSess* packet (see Figure XI-9) carries in its ASP header (contained entirely in the ATP UserBytes) a one-byte *SPCmdType* field = *OpenSess*, a one-byte field containing the WSS socket number, and a two-byte ASP version number field. For this version of the ASP protocol the version number field must be equal to hexadecimal $0100.

Upon receiving an *OpenSess* packet (an ATP transaction request with the *SPCmdType* = *OpenSess*) on an SLS, the server's ASP checks to see if there is an *SPGetSession* pending on that SLS. If no such call is pending then it returns a *ServerBusy* error in the *OpenSessReply* packet and the session is not opened. If such a call is pending then ASP checks the ASP version number in the *OpenSess* packet. If unacceptable, a *BadVersNum* error is returned. If this is acceptable then ASP opens an *ATP-Responding-Socket* SSS and generates a unique (per SLS) one-byte session identifier *SessionID*. Then it creates its internal session management data structures in which the WSS socket number received in the *OpenSess* packet is saved together with the *SessionID*, the SLS socket number, etc. Then the *OpenSessReply* packet is sent back. This contains in its ASP header (contained entirely in the ASP UserBytes) a two-byte *ErrorCode* (returned to the client as *SPError*), the one-byte *SessionID*, and the socket number of the SSS. Now the server end of the session is active. At this time the tickling process is initiated at the server end (details discussed later).

The workstation's ASP upon receiving the *OpenSessReply* examines its *ErrorCode* field. If this indicates *noErr*, then the *SessionID* and the SSS are taken from the packet and together with other control information saved in a session management data structure. At this point, the workstation end of the session is active and the tickling process is initiated at the workstation end (details discussed later).

The session management data structure must contain the session's identifier, the socket number of the other end of the session (i.e. the WSS or the SSS), and a two-byte *LastReqNum*. When the session is opened, the *LastReqNum* is initialized to zero.


## Getting Service Status

Since an *SPGetStatus* call can be made and serviced without opening a session, the corresponding packets do not carry a *SessionID* nor do they have a Sequence number field. The workstation's ASP issues an ATP-ALO transaction request addressed to the SLS. Thus an ATP request known as a *GetStat* packet (see Figure XI-2) is sent to the SLS. This has (see Figure XI-9) the *SPCmdType* = *GetStat* with the rest of the three ATP UserBytes being unused and thus set to zero.

The ASP at the server end upon receiving a *GetStat* packet (i.e. *SPCmdType* = *GetStat*) returns as the possibly multi-packet ATP response up to eight *StatusReply* packets. Each of these has (see Figure XI-10) the four ATP UserBytes equal to zero. The status information block provided in the *SPInit* or *SPNewStatus* call is sent as the ATP data of the *StatusReply* packets. The status information is packed into the packets with as many bytes as will fit (i.e. all but the last *StatusReply* packet will have 578 bytes of the status information in them).

## ASP Commands

When the ASP client in the workstation makes an *SPCommand* call, then ASP sends an ATP-XO request to the SSS of the indicated session (see Figure XI-3). This *CmdReq* packet has (see Figure XI-10) the *SPCmdType = Cmd*, and the *SessID* of the session and a two-byte sequence number. The sequence number must be generated using the following algorithm:

> **If** LastReqNum = 65536 **then** LastReqNum := 0 **else**
>         LastReqNum := LastReqNum+1;
> Sequence Number := LastReqNum;

In effect, the sequence number will be one greater than the sequence number of the last command sent on the session.

The *CmdBlock* provided in the *SPCommand* call is sent in the ATP data part of the *CmdReq* packet. Thus the *CmdBlock* cannot be larger than 578 bytes in size.

At the server end, the ASP upon receiving the *CmdReq* packet delivers it to the ASP client (if there was a pending *SPGetRequest*, otherwise the request is ignored). The ASP client in the server then makes an *SPCmdReply* call with which it passes to ASP a four-byte command result and a variable size command reply data block (of maximum size equal to the *QuantumSize*; for AppleTalk this is 4624 bytes). The ASP generates from one to eight ATP response packets which it sends back to the source of the *CmdReq* packet. These *CmdReply* packets have the four ATP UserBytes set to zero except for the first *CmdReply* which carries the command result in these UserBytes. The command reply data block is broken up into up to eight pieces which are sent in the ATP data part of these packets (see Figure XI-10) packing as many bytes as will fit in each packet (all but the last packet must contain 578 bytes of the command reply block).

## ASP Writes

When the ASP client in the workstation makes an *SPWrite* call, ASP sends an ATP-XO request to the SSS of the indicated session (seeFigure XI-5). This *WriteReq* packet has (see Figure XI-10) the *SPCmdType = Write*, and the *SessID* of the session and a two-byte sequence number. The sequence number must be generated using the algorithm discussed above.

The *CmdBlock* provided in the *SPWrite* call is sent in the ATP data part of the *WriteReq* packet. Thus the *CmdBlock* cannot be larger than 578 bytes in size.

At the server end, the ASP upon receiving the *WriteReq* packet delivers it to the ASP client (if there was a pending *SPGetRequest*, otherwise the request is ignored). The ASP client in the server determines if it can process the request, presumably by examining the contents of the command block.

If the ASP client in the server decides that it is unable to process the request, it encodes an appropriate higher level protocol error message in the four-byte command result and/or command reply data and makes an *SPWrtReply* call to ASP. Then as shown in Figure XI-5, an ATP response packet known as a *WriteReply* is sent back to the source of the *WriteReq*. (see Figure XI-10 for its format).

If however, the ASP client in the server decides that it can process the request, then it reserves a buffer for the data and makes an *SPWrtContinue* call to ASP. This causes the ASP in the server to send an ATP-XO transaction request to the WSS. This *WriteData* call (see Figure XI-11) carries the Session Identifier and the Sequence Number taken from the *Write* packet (used by ASP to match the *WriteData* with the corresponding *Write*). It has a two-byte ATP data field in which the size in bytes of the buffer reserved by the server client for the write data is inserted. The workstation then returns the data in the transaction response packets (*WriteDataReply* packets) to the *WriteData*. At that time the data is delivered to the server-end ASP client. The latter then issues an *SPWrtReply* call to ASP which causes it to send a *WriteReply* packet (this is the ATP response to the original *Write*; see Figure XI-10 for the format of the *WriteReply* packet).

## Reply Size Error Checking

*SPGetStatus, SPCommand* and *SPWrite* all require a buffer into which to put the server's reply, and the length of that buffer. ASP must be able to verify that the entire reply fits into the specified buffer. In general, this information is available from ATP. The workstation-side ASP will always issue an ATP request specifying the number of responses expected as equal to the number of 578 byte blocks that will fit into the caller's buffer, plus one if there is any remainder (buffer size MOD 578 non-zero). There are two cases where size errors could occur that should be examined.

First, the server's response is of a size such that the number of response buffers the server-side ASP wishes to return is equal to the number requested, but the response still is too big to fit in the caller's buffer. In this case the workstation ASP can determine there was an error because ATP will indicate that the last response did not fit in the last response buffer.

Second, the server's response is of a size such that the number of response buffers the server-size ASP wishes to return is greater than the number requested by the workstation side. In this case, the server side should return as much of the response as it can (i.e. completely fill each requested response, including the last one, with 578 bytes of data). As long as the caller's buffer size was not an exact multiple of 578 bytes, the workstation ASP can again determine a size error occurred because the last response again will not fit in the last response buffer.

However, if the caller's buffer size was an exact multiple of 578 bytes, ASP can not determine, solely on the basis of information returned by ATP, if a size error occurred. If the last response buffer is completely filled with data, it could be either because the response was exactly the requested size, or the response was greater than the requested size but the server side had no way of passing this information back to the workstation. To be able to resolve this situation, the workstation-side ASP must ask for one additional response in this case. If the response size is less than or equal to the caller's buffer size, this response will not be returned (the EOM bit will be set). However, if the response size is greater than the caller's buffer size, this additional response will be returned, making the workstation-side ASP aware of the error. Note that if the caller's buffer is exactly *QuantumSize* big (which is a multiple of 578), this situation does not apply, since this is the most data that can possibly be sent by the server side.

## Tickles and Session Maintenance

*Tickle* packets (these are ATP TReq packets with *SPCmdType* = *Tickle* ) must be sent by each end while a session is open (see figure XI-4). The tickle packets are sent by the

workstation to the SLS and by the server to the WSS. Tickle packets contain in the ASP header (the ATP UserBytes) the one byte *SPCmdType = Tickle*, the *SessionID*, and two unused bytes (see figure XI-9). Tickle packets are sent by starting an ATP transaction with retry count equal to infinite and time-out equal to 30 seconds.

The session maintenance at each end is done by starting a *session maintenance timeout* of 2 minutes. Whenever any packet (tickle or otherwise) is received on the session this timer is restarted. If the timer expires (i.e. if no packet is received for 2 minutes) then it is concluded that the other end of the session has "died" or has become unreachable, and the session is closed.

Attention Requests

When the ASP client in the server makes an *SPAttention* call, ASP sends an ATP-ALO request to the WSS of the indicated session (see figure XI-8). This *Attention* packet requests one response buffer and has (see figure XI-9) the *SPCmdType = Attention*, the *SessID* of the session and one word (two bytes) of *attention data* . This attention data is passed by the server client to ASP to be delivered to the workstation client along with the attention request. It is uninterpreted by ASP except for the fact that ASP requires it to be non-zero.

The workstation-side ASP, upon receiving an *Attention* request, should immediately respond with an *AttentionReply* . This serves as an acknowledgement of the request, and completes the *SPAttention* call on the server side. The workstation ASP should then, in an implementation-dependent manner, alert its client as to the attention request and pass the client the attention data.

Closing a Session

When the ASP client at either end makes an *SPCloseSession* call, a session closing ATP-ALO transaction is initiated. (see Figures XI-6 and XI-7).

If the session closing was initiated by the workstation client, then the *CloseSess* packet (an ATP transaction request) is sent to the SSS in the server, where it is delivered to the ASP client as part of the *SPGetRequest* mechanism. The server's ASP generates the TResp -- a *CloseSessReply* packet -- without the ASP client's intervention. Immediately upon sending the *CloseSessReply* packet the server end of the session is closed, all pending activity including tickles is cancelled at that end, and no further server end calls to this session are accepted. The workstation end, immediately upon receiving the SPCloseSession call, cancels all pending activity on the session including tickles, and does not accept any more calls from its ASP client referring to that session. The workstation end can choose (this is implementation dependent) to retransmit the *CloseSess* packet (remember, it's an ATP transaction request) several times; it will close its end of the session either as soon as the *CloseSessReply* is received or the retries expire.

It is possible that on the server side no *SPGetRequest* was pending when the close request was received. In this case, everything should proceed as above and the session should be marked as closed. The server client, however, can not be informed until his next request for that session. At this time, ASP should return a *SessClosed* or other error.

If the session closing was initiated by the server end then the *CloseSess* is sent to the WSS. The workstation's ASP generates the *CloseSessReply* without client intervention. The

session is closed immediately upon receipt of the *CloseSess* packet. The ASP client in the workstation is informed of the session being closed when it makes a call to ASP and refers to that session. The server end can choose (this is implementation dependent) to retransmit the *CloseSess* packet several times; it will close its end of the session either as soon as the *CloseSessReply* is received or the retries expire.

The formats of the *CloseSess* and *CloseSessReply* packets are given in Figure XI-9. Note that the *CloseSess* packet does not include a sequence number and hence must be accepted by the receiving end without sequence number verification. Also, its receipt should immediately lead to the cancellation of all pending activity on the session, including tickles.

*CloseSess* packets are sent simply as a courtesy to the other end, and to potentially speed up the process of closing a session at both ends. Note that it is possible that the *CloseSess* packet or the *CloseSessReply* sent by either end can get lost; then the end initiating the session closing activity (let's assume it has chosen to send the *CloseSess* packet only once) will not receive an acknowledgement to its request and will close the session and stop tickles when its retries expire. If the other end in fact did not receive the *CloseSess* packet at all, it will not know that the session has been closed. However, this half-open session will be detected when the still active end's session maintenance timer expires, at which time it will close its own end of the half-open session.

## SPCmdType Values

The standard values of *SPCmdType* are as follows:

> *CloseSession* = 1
> *Command*= 2
> *GetStat* = 3
> *OpenSess* = 4
> *Tickle* = 5
> *Write* = 6
> *WriteData* = 7
> *Attention*= 8.

All other values are invalid in this field.

## SPError Values

The standard values of *SPError* are as follows:

| Error | Hex Value | Decimal Value | Where Returned |
|---|---|---|---|
| NoError | 0 | 0 | Both sides (*) |
| BadVersNum | $FBD6 | -1066 | Workstation (*) |
| BufTooSmall | $FBD5 | -1067 | Workstatioon |
| NoMoreSessions | $FBD4 | -1068 | Both sides |
| NoServers | $FBD3 | -1069 | Workstation |
| ParamErr | $FBD2 | -1070 | Both sides |
| ServerBusy | $FBD1 | -1071 | Workstation (*) |
| SessClosed | $FBD0 | -1072 | Both sides |
| SizeErr | $FBCF | -1073 | Both sides |
| TooManyClients | $FBCE | -1074 | Server |
| NoAck | $FBCD | -1075 | Server |

Values -1060 through -1065 are reserved for implementation-dependent errors. All other values are invalid in this field. Error codes marked with a (\*) are the only ones actually transmitted through ATP (on the OpenSession call).

## Time-Outs and Retry Counts

ASP uses ATP transactions as its basic building blocks. For each of these transactions it is necessary to provide a retransmission time-out value and maximum retry count.

Most transactions used by ASP (except: opening a session, getting service status, requesting attention and closing a session) use a retry count of infinite. This involves no danger of leading to a deadlock, since half-open connections (other end "dead" or unreachable) are easily detected through the tickling mechanism.

The maximum number of retries used by the session opening, getting service information and attention transactions should be specifiable by the ASP user. How this is done is an implementation issue not discussed in this document.

The time-out value to be used in any of the transactions (with the exception of tickles) and how this is specified by the user or built into ASP is an implementation issue and is not specified in this document.

WorkStation                          Server

**SPOpenSession** →            OpenSess

                        TReq(TID = t1)           SLS

                                                 The OpenSessReply is
                                                 sent back by ASP
                                                 without its client's
                                                 intervention

                            OpenSessReply        SLS

SPOpenSession results ←
returned to ASP client          TResp(TID = t1)

         time                       time

**Figure XI-1. Session opening transaction**


WorkStation                          Server

**SPGetStatus** →              GetStat

                        TReq(TID = t2)           SLS

                                                 The StatusRpiy is
                                                 sent back by ASP
                                                 without its client's
                                                 intervention

                            StatusReplys         SLS

                                    .

SPGetStatus results ←
returned to ASP client          TResps(TID = t2)

         time                       time

**Figure XI-2. Get Status transaction**

**Figure XI-3. Command transaction**



**Figure XI-4. Tickling transactions**

WorkStation                                    Server

SPWrite ──────►│       Write                │
               │                            │ SSS ────► delivered to ASP client
               │  TReq(TID = t)             │
               │                            │         ASP Client Error
               │                            │
               │       WriteReplys          │◄────── SPWrtReply
               │◄────────── •               │
SPWrite results returned ◄──│◄────────      │
to ASP client               │   TResps(TID = t)

SPWrite ──────►│       Write                │
               │                            │ SSS ────► delivered to ASP client
               │  TReq(TID = t1)            │
               │       WriteData            │◄────── SPWrtContinue
      WSS ◄────│    TReq(TID = t2)          │
               │    WriteDataReplys         │
               │──────────────►             │
               │          •                 │
               │──────────────►             │
               │  TResps(TID = t2)     ────► Write data delivered
               │                            │         to ASP client
               │       WriteReplys          │ SSS ◄────── SPWrtReply
               │◄────────── •               │
SPWrite results returned ◄──│◄────────      │
to ASP client               │   TResp(TID = t1)

                time                        time

Figure XI-5. Write transaction

WorkStation                    Server

SPCloseSession   →      CloseSess
                        TReq(TID = t2)
                                              delivered to ASP client
                                         SSS

the session is closed at the    CloseSessReply
workstation end as soon as       TResp(TID = t2)              session closed at
the CloseSessReply is                                         server end
received or the retries expire,
whichever occurs first.
                                                    ASP generates the
                                                    CloseSessReply
                                                    without the ASP
                                                    client's intervention

                   time                      time
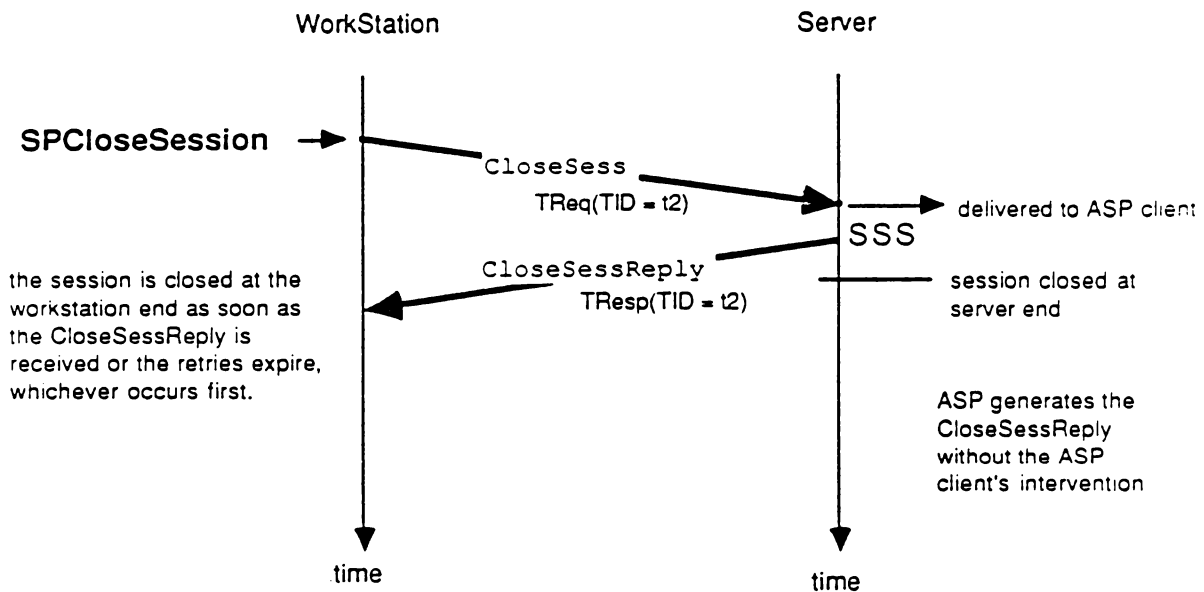
Figure XI-6.  Session closing transaction (workstation initiated)


WorkStation                    Server

                          CloseSess           ←   SPCloseSession
ASP generates the         TReq(TID = t3)
CloseSessReply      WSS ◄
without the ASP
client's intervention                         the session is closed at the
                          CloseSessReply      server end as soon as
                          TResp(TID = t3)     the CloseSessReply is
                                              received or the retries expire.
     session closed at                        whichever occurs first.
     workstation end

                   time                      time
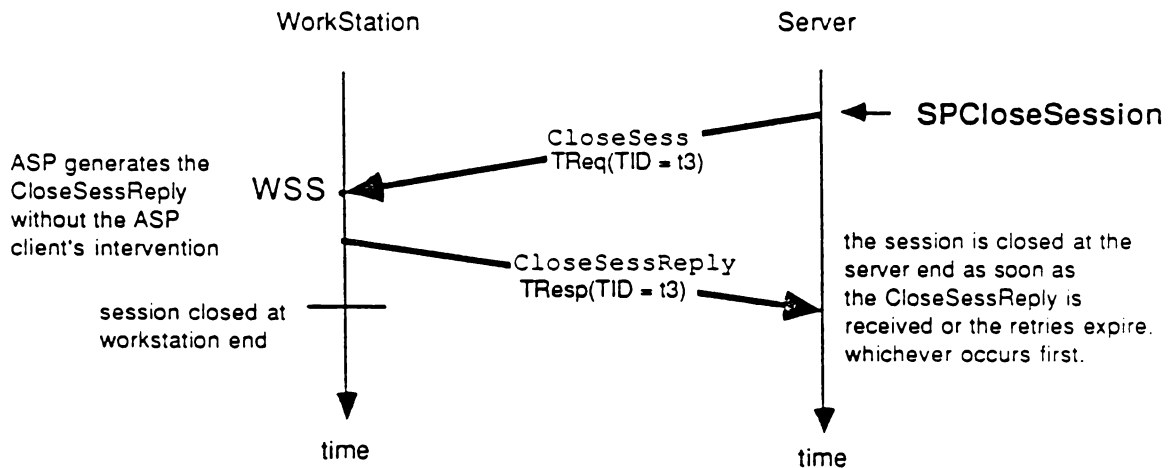
Figure XI-7.  Session closing transaction (server initiated)
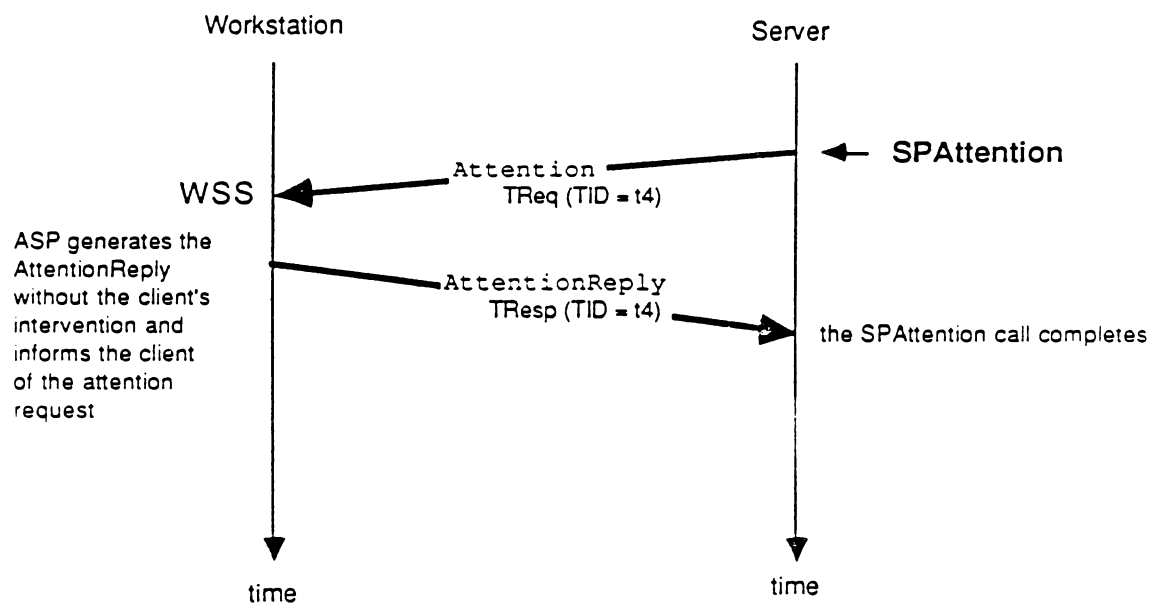
Figure XI-8. Attention request

1 byte

LAP header

DDP header

ATP Header

ATP User Bytes

| OpenSess packet | OpenSessReply packet | Tickle packet | CloseSess packet |
|---|---|---|---|
| SPCmdType = OpenSess | SSS number | SPCmdType = Tickle | SPCmdType = CloseSess |
| WSS number | SessID | SessID | SessID |
| SP Version number | Error Code | 0 | 0 |

hi byte

lo byte

1 byte

LAP header

DDP header
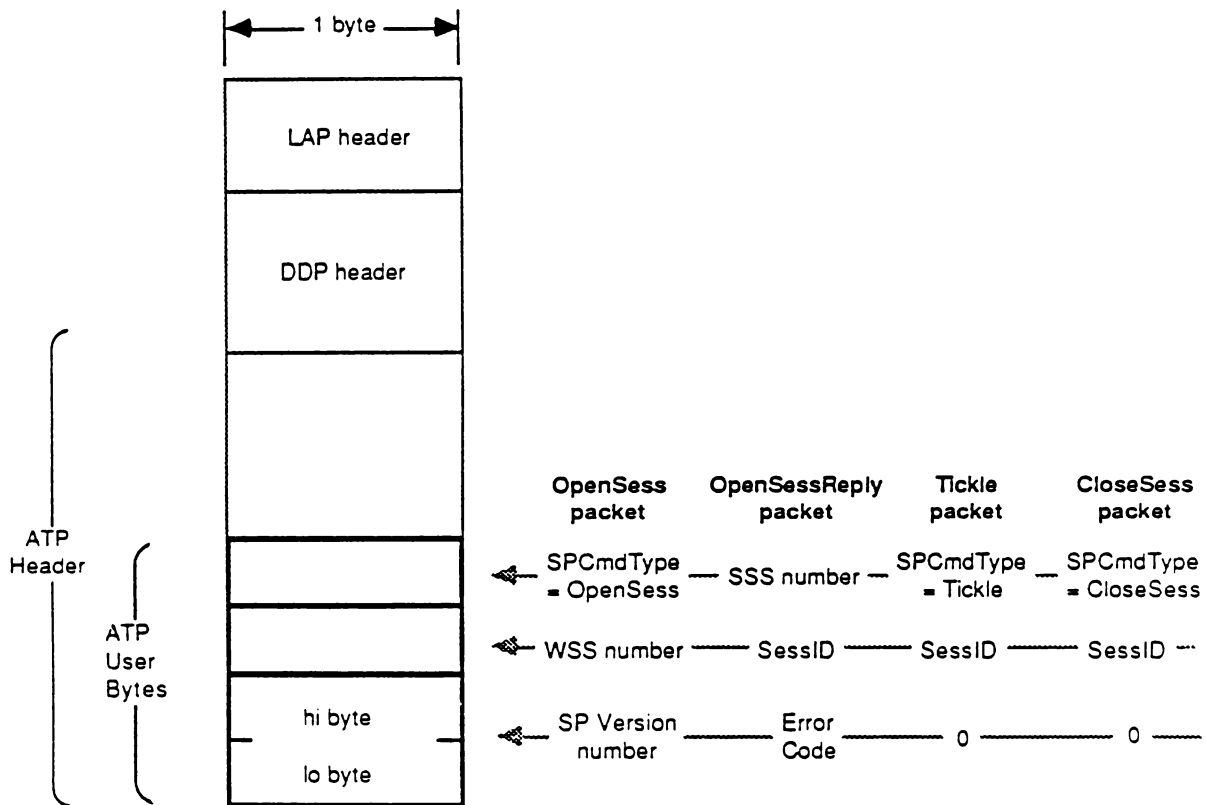
**Figure XI-9. Packet Formats**

ATP Header

ATP User Bytes

| GetStat packet | CloseSessReply packet | Attention packet | AttentionReply packet |
|---|---|---|---|
| SPCmdType = GetStat | 0 | SPCmdType = Attention | 0 |
| 0 | 0 | SessID | 0 |
| 0 | 0 | Attention Word | 0 |

hi byte

lo byte

1 byte

LAP header

DDP header

ATP Header

ATP User Bytes

hi byte

lo byte

ATP Data (0..578 bytes)

| Command packet | Write packet | StatusReply packet | CommandReply and WriteReply packet | WriteDataReply packet |
|---|---|---|---|---|
| SPCmdType = Command | SPCmdType = Write | 0 | CmdResult* (byte 1) | 0 |
| SessID | SessID | 0 | CmdResult* (byte 2) | 0 |
| Sequence Number | Sequence Number | 0 | CmdResult* (bytes 3 and 4) | 0 |
| Command Block | Command Block | Status Block | Command Reply Data | Write Data |

* These fields must be equal to zero except in the first response packet of the transaction where they carry the command result.

**Figure XI-10. Packet Formats (continued)**

```
          |◄—— 1 byte ——►|

         ┌───────────────┐
         │               │
         │  LAP header   │
         │               │
         ├───────────────┤
         │               │
         │  DDP header   │
         │               │
         ├───────────────┤
         │               │
         │               │
   ⎧     │               │              WriteData
   │     │               │               packet
 ATP│    ├───────────────┤
Header   │               │ ◄──────────  SPCmdType _____
   │ ⎧   │               │              = WriteData
   │ │   ├───────────────┤
   │ ATP │               │ ◄──────────── SessID ─────────
   │User │               │
   │Bytes├───────────────┤
   │ │   │   hi byte     │ ◄──────────  Sequence _____
   │ │   ├───────────────┤              Number
   ⎩ ⎩   │   lo byte     │
         ├───────────────┤
  ATP    │   hi byte     │ ◄──────────  Available Buffer _____
  Data   ├───────────────┤              Size
(2 bytes)│   lo byte     │
         └───────────────┘
```

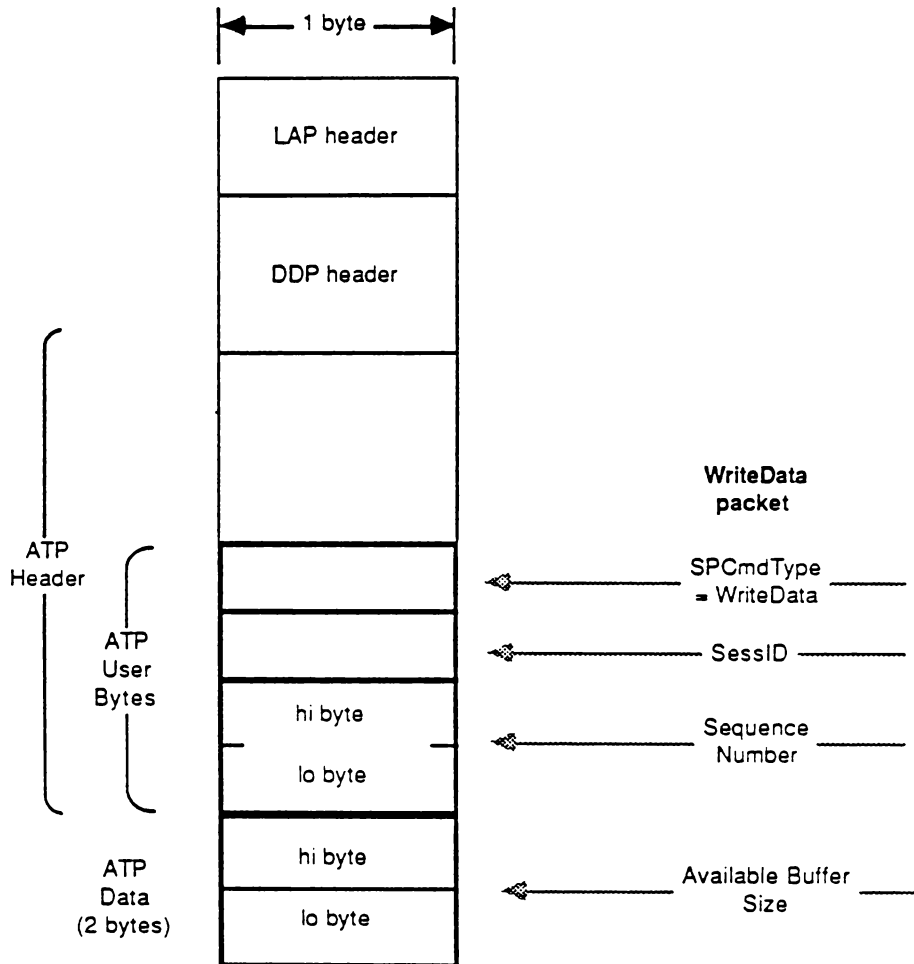**Figure XI-11.  ASP Packet Formats (continued)**

# Appendix A

# AppleTalk Electrical/Mechanical Specifications

This appendix reproduces the two relevant Apple Computer specification documents for AppelTalk's electrical and physical aspects.

The first document (Apple document number 062-0190-B) is titled "AppleBus Electrical/Mechanical Specification." The use of the name "AppleBus" is historical and should be read in this context as "AppleTalk." This document provides the detailed electrical specifications as well as cable and connector characteristics.

The second document (Apple document number 157-0049-C) is titled "Transformer Specification." It provides the detailed specification of the transformer used in the AppleTalk connection module.

These documents are reproduced here for the convenience of developers wishing to implement AppleTalk compatible connections on their devices.

# AppleBus Electrical/Mechanical Specification

| ⊕ ◁ | METRIC | apple computer inc. |
|-----|--------|---------------------|

DIMENSIONS ARE IN MILLIMETERS
TOLERANCES

| | X | ANGLES |
| XX | | |

MATERIAL | FINISH

| DR | GARDNER | B/85 | DR | C/84 |
| ENG APPD | 22 | | | |
| RELEASE | | /84 | | |

TITLE

SPECIFICATION, APPLEBUS
ELECTRICAL/MECHANICAL

| DESIGN NUM | SCALE | SIZE | DRAWING NUMBER | SH |
|-----|-----|------|----------------|----|
| — | — | A | 062-0190-B | 1/10 |

## 1.0 Introduction

AppleBus is a serial interconnection system for all Apple Computers and several future peripheral products. It provides a common, convenient, inexpensive, expansion capability for computer products. In summary, AppleBus is a multi-drop, balanced, transformer isolated, serial communication system designed to connect 32 devices at 230.4 Kbaud over a total distance of up to 300 meters.

## 2.0 References

AppleBus Link Access Protocol, specification number 062-0214.

EIA Standard RS-422, Electrical Characteristics of Balanced Voltage Digital Interface Circuits.
Fairchild Semiconductor "Interface, Line Drivers and Receivers", 1975.

## 3.0 Summary of Features and Performance

* 32 total devices

* Shielded, twisted pair, connectorized interconnection; easy user configuration, maximum total cable length of 300 meters

* 230.4 Kbaud communication, SDLC frame format, FM0 modulation

* Balanced signalling using standard RS-422 driver (26LS30) and receiver (26LS32) I.C.'s

* Transformer isolation for excellent noise and static discharge immunity

* Self-configuring, no user switches or action to identify devices

* Passive drops, devices may be disconnected, and one may fail without disturbing communication

---

**apple computer inc.**

| SIZE | DRAWING NUMBER |
|------|----------------|
| A | 062-0190-3 |

| SCALE ——— | SHEET 2 OF 10 |

## 4.0 Signalling

4.1 AppleBus devices send and receive data over a single pair of wires connected to each device. Two connectors on each device allow the user to easily connect devices together by means of a simple cable (see Section 5.0 Interconnection). Balanced, transformer coupled signalling is used to reduce both RFI and noise susceptibility.

4.2 Each device has a single driver and receiver. The receivers are always connected to the system and pass all AppleBus data to the controller. Only one driver at a time is enabled. Software controls which device may transmit data (see Protocol Specification). The driver and receiver descriptions and specifications are given in the electrical section, 6.0.

4.3 The signal on AppleBus is encoded with FM0 (bi—phase space). This insures that clock information can be recovered at the receivers. In FM0, a transition occurs at the beginning of every bit cell. A "0" is represented by an additional transition at the center of the bit cell, and a "1" is represented by no transition at the center of the cell.



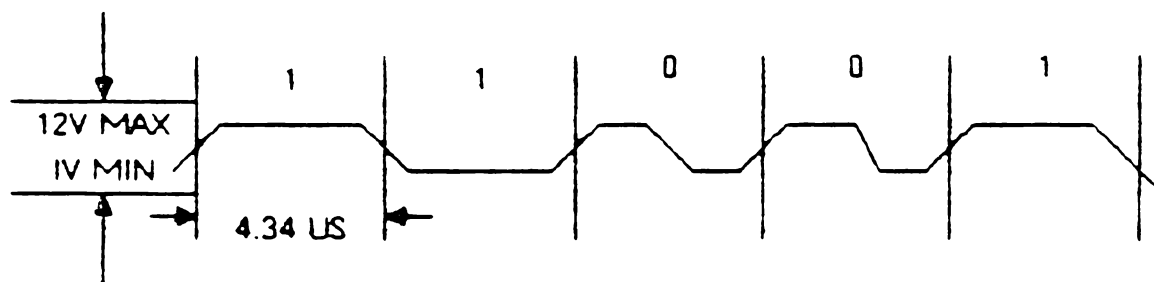Figure 4.1 FM0 Coding (230.4 K baud +/ - 1%)

4.4 Synchronization time for the clock recovery system is provided by the transmission of 14 consecutive 1 bits directly preceding the data frame. Frame format is covered in the Protocol Specification.

## 5.0 Interconnection

5.1 Applebus devices are connected to the AppleBus by a connection module
which contains a transformer, DB-9 connector at the end of an 460 millimeter
cable, and two 3-pin miniature DIN connectors as shown in Figure 5.1. Each
3-pin connector has a coupled switch. If both connectors are used, the
switches are open, but if one of the conectors is not used, a 100 ohm
termination resistor (R2) is connected across the line. The use of the connection
module allows the AppleBus device to be removed from the system by
disconnecting it from the module without disturbing the operation of the bus.
R3 and R4 increase the noise immunity of the receivers, while R5 and C1
isolate the frame grounds of the AppleBus devices and prevent ground loop
currents. The resistor (R1) provides static drain for the cable shield to
ground.
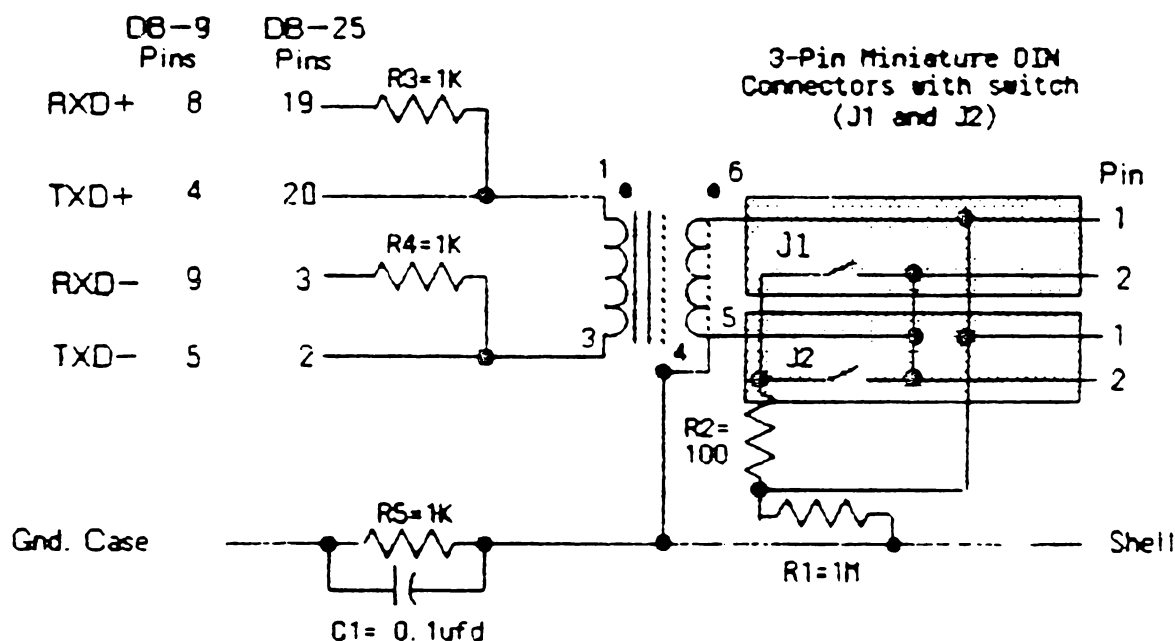
Figure 5.1  AppleBus Connection Module

5.2 Individual AppleBus devices are connected together by a twisted, shielded pair cable. The cable is available in standard lengths with connectors on each end, or in 150 meter bulk rolls. The maximum length of cable for the bus is limited to a total of 300 meters.

Cable Specification

| | |
|---|---|
| Conductors: | 22 AWG stranded  17 ohm per 300 meters |
| Shield: | 85% coverage braid |
| Impedance: | 78 ohm |
| Capacitance: | 68 pF per meter |
| Rise Time: | 175 ns 0 to 50% at 300 meters |
| Diameter: | 4.7 mm (0.185 inches) maximum |

5.3 The AppleBus connector is a miniature 3 pin circular connector similar to Hosiden connector number TCP8030-01-010.

Pin 1   AppleBus  —  Plus

Pin 2   AppleBus  —  Minus

Pin 3   Unused

Shell   Shield
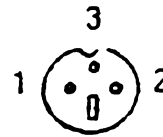
Figure 5.2  Connector Pin Assignment (looking into connector)

5.4 The interconnecting cable is wired "one-to-one" as shown below.
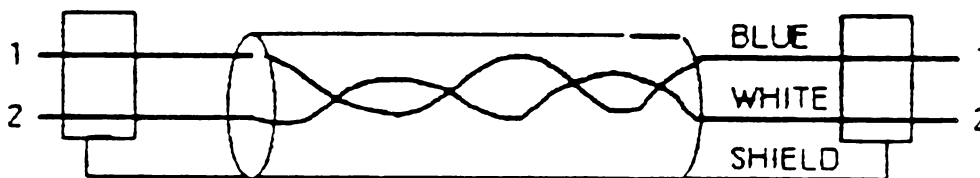
Figure 5.3  Interconnecting Cable Connection

## 6.0 Electrical Specification

6.1 The recommended driver is the 26LS30 used with both +5V and −5V as power supplies, and the mode control connected to give differential outputs (Mode voltage low).

The outputs of the driver are coupled to the connector through a "deglitch network" which consists of a T−network of two 20 to 30 ohm resistors and a 150 to 300 picofarad capacitor. Use of the network gives two major advantages. The first is that the high frequency components of the signal are attenuated both going onto and off of the cable thus reducing RFI and also static susceptibility. The second advantage is that at least one driver on the network can fail without causing the network to fail (as long as it fails in one state and doesn't broadcast trash).

Those who wish to use standard RS−422 specified components (power supplies of +5 volts and ground) may do so if the deglitch circuits are not used, but should be aware that the driver must drive a cable impedance of 39 ohms (middle of a 78 ohm cable).

6.2 The receiver is the 26LS32. Both inputs of the receiver are connected through deglitch networks.



Figure 6.1   Driver and Receiver Connection

| | SIZE | DRAWING NUMBER |
|---|---|---|
| apple computer inc. | A | 062-0190-B |
| | SCALE: ——— | SHEET 7 OF 10 |

6.3 The transformer is a 1:1 turns ratio transformer with tight coupling between primary and secondary, and electrostatic shielding to give excellent common mode isolation.

The primary is wound as two windings of #32 AWG wire in series with one wound below the secondary and one above it.



P.C. PATTERN

Dimensions in Inches
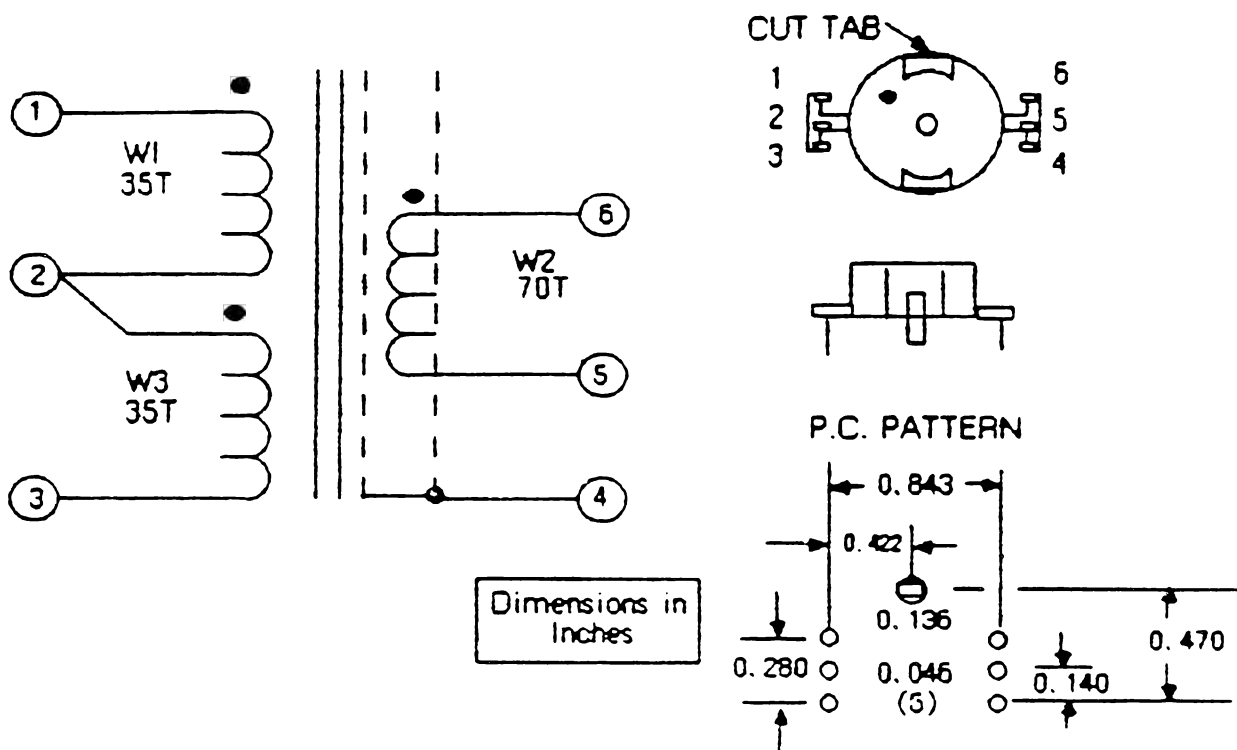
Specifications:

| | |
|---|---|
| Core Material: | Siemans B65651-K000-R030 (or equivalent) |
| Bobbin: | Siemans B65652-PC1,L (or equivalent) |
| Retaining Clip: | Siemans B65653-T (or equivalent) |
| Magnetizing Inductance: | 20 mH minimum |
| Leakage Inductance: | 15 uH max |
| Capacitance: | 5 pF max (primary to secondary with electrostatic shield and core guarded) |

Figure 6.2  Transformer Specification

apple computer inc.

| SIZE | DRAWING NUMBER |
|---|---|
| A | 062-0190-B |

| SCALE: ——— | SHEET 8 OF 10 |

6.4 Each end of the cable must be terminated. The AppleBus connection module is designed such that a 100 ohm resistor is connected across the line if one of the two connectors is not used. A 100 ohm resistor is used even though the characteristic impedance of the line is 78 ohms because it gives adequate termination and minimizes resistive losses.

6.5 The cable shield should be grounded to Earth ground (frame ground) at each AppleBus device. This ground is necessary to prevent excessive RFI. A resistor and capacitor are included in each connection module to isolate the cable shield from the ground connection at 60 Hz while offering a low impedance connection to high frequency noise. This connection scheme allows ground connection for RFI purposes without risking high currents flowing in the cable due to differing ground potentials. In the U.S., the green wire available in most outlets is suitable for frame grounding.
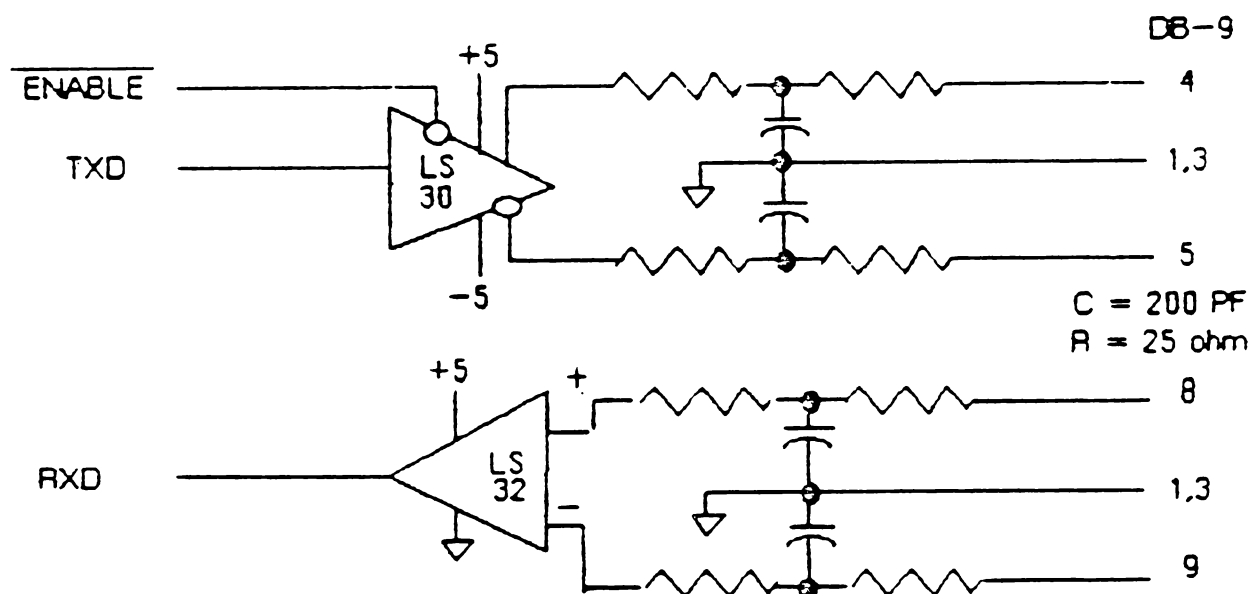
<table>
<tr><td rowspan="2">apple computer inc.</td><td>SIZE<br>A</td><td colspan="2">DRAWING NUMBER<br>062-0190-B</td></tr>
<tr><td>SCALE: ————</td><td>SHEET 9</td><td>OF 10</td></tr>
</table>

## 7.0 Documentation

The following is a list of all AppleBus hardware drawings.

| Drawing Description | Drawing Number |
|---|---|
| AppleBus Connection Module Cover (Top) | 815-0839 |
| AppleBus Connection Module Cover (Bottom) | 815-0838 |
| AppleBus Cable Extender (Coupler) | 519-0300 |
| AppleBus Cable with Molded Plugs | 590-025X |
| DB-9 Connection Cable Assembly | 590-0254 |
| DB-25 Connection Cable Assembly | 590-0253 |
| AppleBus Board Film Artwork | 820-0135 |
| AppleBus Connection Module FCC Label | 825-1008 |

# Transformer Specification

METRIC

 apple computer inc.

TITLE

Transformer

| G. Crow | | A | 157-0049-C | 1/5 |

157-0049-C

1/5

## 1.0 Description

This transformer is used in the Appletalk connection module to give isolation between the Appletalk cable and the devices which are connected to the cable.

The transformer is a 1:1 turns ratio transformer with tight coupling between primary and secondary, and electrostatic shielding to give excellent common mode isolation.

The primary is wound as two windings of #32 AWG wire in series with one wound below the secondary and one above it. The secondary is a single continuous winding of #32 wire.

## 2.0 Environmental

The transformer shall operate properly and meet its specifications under the following environmental conditions:

| | |
|---|---|
| Operating Temperature: | 0 to 70 ° C |
| Storage Temperature: | -40 to 70 ° C |
| Relative Humidity: | 5 to 95 % |
| Altitude: | 0 to 4572 meters |

In addition, the transformer must meet the Apple Computer shock and vibration requirements while mounted on a printed circuit board and tested to Apple specification number 062-0086.

## 3.0 Mechanical Strength and Workmanship

The transformer winding assembly, pins, mounting plate, core, and clamp shall be securely mounted and rigid with respect to each other.

The pins must be easily solderable; solderability must meet EIA-RS-186-9E.
All components shall be free of undue mechanical stresses.

## 4.0 Identification Markings

The transformer shall be marked with the manufacturer's name or identification number, date of manufacture, country of origin, manufacturer's part number, and the Apple part number. The markings shall be clearly legible after typical assembly, wave-soldering, and cleaning processes, and after exposure to the above mentioned environmental extremes. The markings may be placed in any convenient and visible location on the transformer.

| apple computer inc. | SIZE | DRAWING NUMBER |
|---|---|---|
| | A | 157-0049-C |
| | SCALE: | SHEET 2 OF 5 |

## 5.0 Electrical Specifications

| | |
|---|---|
| Core Material: | Siemans B65651-K000-R030 (or equivalent) |
| Bobbin: | Siemans B65652-PC1,L (or equivalent) |
| Retaining Clip: | Siemans B65653-T (or equivalent) with washer if required |
| Magnetizing Inductance: | 15 mH minimum @ 10 KHz, 1 V RMS measured between pins 1-3 |
| Leakage Inductance: | 17 $\mu$H max @ 10 KHz (measured between pins 1-3 with pins 5-6 shorted.) |
| Resistance: | 1.5$\Omega$ max (measured between pins 1-3, and 5-6) |
| Capacitance: | 6 pF max (primary to secondary with electrostatic shield and core guarded) |
| Turns Ratio: | 1:1 accurate to the nearest 1/2 turn |
| Hi Pot: | From W2 to core, shield and primary 1000 VDC for one minute with no significant leakage current. |
| Magnitude of Impedance (IZI) | 10K$\Omega$ minimum @ 250 KHZ Measured with HP 4800A Vector Impedance Meter between pins 5-6 |

Notes:    Different permeablility core material may be used, and turn count may be modified as long as all electrical specifications are met.

All wire #32 AWG.
Turns counts are typical and may be adjusted
for different core materials.
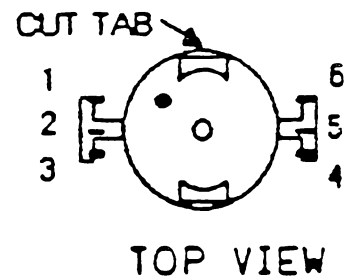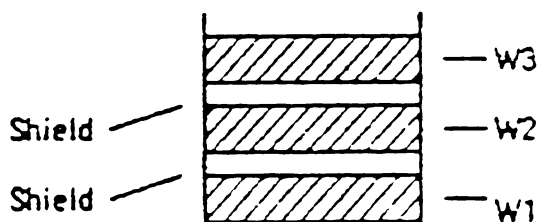
Figure 1.0   Schematic and Build Detail

18.2 REF

16.5, MAX

4.7 ± 0.7, 7 PL

1.7 MAX

1.7 MAX

TAB MATERIAL
0.5 THK MAX

0.35 X 0.70, 6PL
(OPTIONAL ∅ 0.60)

3.56

8.35

19.5
MAX

7.12 ON ₵

PIN 1
REF

21.54
ON ₵

CUT TAB

28.5 MAX
ON ₵

PIN 1
REF

21.54

∅ 1.17
6 PL

7.12

10.72

3.56

12.35

NOTES:
1. TRANSFORMER TO FIT EASILY INTO
   P.C. HOLE PATTERN.

2. UNLESS NOTED, 0.XX +/-0.20%

1.27
ON ₵

3.56 ON ₵
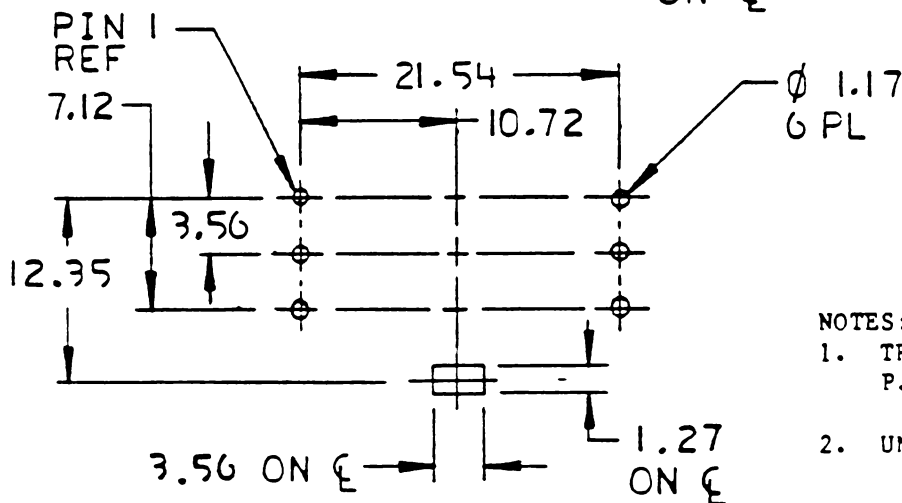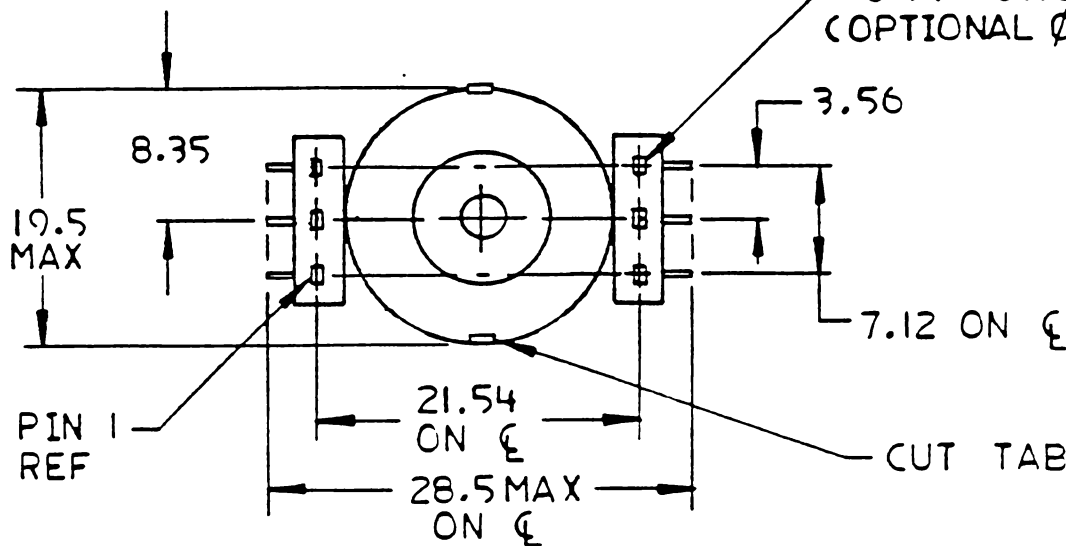
PCB HOLE PATTERN
COMPONENT SIDE

FIGURE 2.0 MECHANICAL SPECIFICATION

apple computer inc.

| SIZE A | DRAWING NUMBER 157-0049-C |
| SCALE 2X | SHEET 5 OF 5 |

# Appendix B

# Miscellaneous AppleTalk Parameters

This appendix summarizes various numerical quantities used in the AppleTalk protocols. It provides a single reference source for this information. This information is organized into subsections, one for each relevant protocol.

The notation used in this section is: $.... represents hexadecimal, %.... represents binary, all other numbers are decimal.

### ALAP Parameters

ALAP protocol type field values:

| | |
|---|---|
| $00 - | invalid ALAP protocol type value (not to be used) |
| $01 through $7F - | valid ALAP protocol type values for use in ALAP client packets; |
| $01 through $0F - | reserved for Apple Computer's use only |
| $01 - | DDP short header packet |
| $02 - | DDP long header packet |
| $0F - | experimental ALAP packet (used by Apple only) |
| $80 through $FF - | reserved for ALAP control frames |
| $81 - | ALAP ENQ packet |
| $82 - | ALAP ACK packet |
| $84 - | ALAP RTS packet |
| $85 - | ALAP CTS packet |

This information is summarized pictorially in Figure B-1.

Timing constants used by ALAP:

There are three constants used by ALAP:

| | |
|---|---|
| Inter-Frame Gap - | less than 200 microseconds |
| Inter-Dialogue Gap - | at least 400 microseconds |
| IDG slot - | 100 microseconds |

ALAP frame parameters:

Flag byte used for framing an ALAP packet = %01111110
Number of flag bytes needed at the start of frame = two or more
Number of bits in abort sequence - 12 to 18
Maximum number of data bytes in ALAP frame (not including headers, etc.) = 600

## DDP Parameters

<u>DDP packet parameters:</u>

ALAP protocol type value for short header DDP packet = 1
ALAP protocol type value for long header DDP packet = 2
Maximum number of data bytes in a DDP packet = 586 bytes

<u>DDP protocol type field values:</u>

| | |
|---|---|
| $00 - | invalid DDP protocol type value (not to be used) |
| $01 through $FF - | valid DDP protocol type values for use in DDP client packets; |
| $01 through $0F - | reserved for Apple Computer's use only |
| $01 - | RTMP response or data packet |
| $02 - | NBP packet |
| $03 - | ATP packet |
| $04 - | EP packet |
| $05 - | RTMP request packet |
| $06 - | ZIP packet |
| $07 - | ADSP packet |

This information is summarized pictorially in Figure B-2.

<u>Socket numbers:</u>

| | |
|---|---|
| $00 - | invalid (do not use) |
| $FF - | invalid (do not use) |
| $01 through $FE - | valid DDP sockets |
| $01 through $7F - | statically assigned sockets |
| $01 through $3F - | reserved for Apple Computer's use only |
| $40 through $7F - | experimental use only (<u>not to be used in released products</u>) |
| $01 - | RTMP socket |
| $02 - | Names Information Socket |
| $04 - | Echoer Socket |
| $06 - | Zone Information Socket |

This information is summarized pictorially in Figure B-3.

## RTMP parameters

RTMP Socket = socket number 1
Maximum number of hops = 16

<u>RTMP packet parameters:</u>

DDP protocol type value RTMP response or data packets = 1
DDP protocol type value RTMP request packets = 5
RTMP request packet comamnd field values:
        1 = Request

RTMP Timers:

    Send-RTMP Timer = 10 seconds
    Validity Timer = 20 seconds
    Timer for aging A-Bridge in a non-bridge node = 90 seconds

## NBP parameters

    Names Information Socket = socket number 2
    Maximum number of characters in object, type or zone fields = 32

Wildcards in NBP names:

    *      used only in the zone field of an entity name to mean the zone of the
               packet's sender,
    =      used as the object and/or type field of an entity name to mean all objects
               and/or all types (can not be used as the zone field)

NBP packet parameters:

    DDP protocol type value for NBP packets = 2
    NBP Control field values:
        1 = BrRq
        2 = LkUp
        3 = LkUp-Reply

## ATP parameters

ATP packet parameters:

    DDP protocol type value for ATP packets = 3
    Function code values:
        %01 = TReq
        %10 = TResp
        %11 = TRel
    Maximum size of data in an ATP packet = 578 bytes

ATP Timers:

    Release timer = 30 seconds

## ZIP parameters

    Zone Information Socket = socket number 6

ZIP packet parameters:

    DDP protocol type value for ZIP packets = 6  (except GetMyZone and GetZoneList
                                               packets)
    ZIP Command values:

```
1 = ZIP query
2 = ZIP reply
3 = ZIP takedown
4 = ZIP bringup
7 = ZIP GetMyZone
8 = ZIP GetZoneList
```

## ZIP Timers:

Query retransmission timer = 10 seconds
ZIP bringback timer = 10 minutes

## PAP parameters

### PAP packet parameters:

Maximum data size in packet = 512 bytes
PAP Type values:
```
1 = OpenConn
2 = OpenConnReply
3 = SendData
4 = Data
5 = Tickle
6 = CloseConn
7 = CloseConnReply
8 = SendStatus
9 = StatusReply
```
Maximum length of status string = 255 bytes (not including the length byte)

### PAP Timers and Retry Counts:

OpenConn request ATP retry timer = 2 seconds
OpenConn request retry count = 5
Tickle timer = 60 seconds
Connection timer = 2 minutes
SendData request retry timer = 15 seconds

## EP parameters

Echoer Socket = socket number 4

### EP packet parameters:

DDP protocol type value for EP packets = 4
Echo Function values:
```
1 = Echo request
2 = Echo reply
```
Maximum data size = 585 bytes

## ASP parameters

<u>ASP packet parameters:</u>

SPCmdType values:
1 = CloseSession
2 = Command
3 = GetStat
4 = OpenSess
5 = Tickle
6 = Write
7 = WriteData
8 = Attention

<u>ASP Timers:</u>

Tickle timer = 30 seconds
Session maintenance timer = 2 minutes

**Valid LAP protocol types**

0 — 15 — 127 — Reserved for LAP frames — 255

| 0 Invalid | 1 DDP Short | 2 DDP Long | 3-14 Reserved for Apple | 15 |
|---|---|---|---|---|

Experimental

| $80 | $81 ENQ | $82 ACK | $83 | $84 RTS | $85 CTS | $86-$FF |
|---|---|---|---|---|---|---|
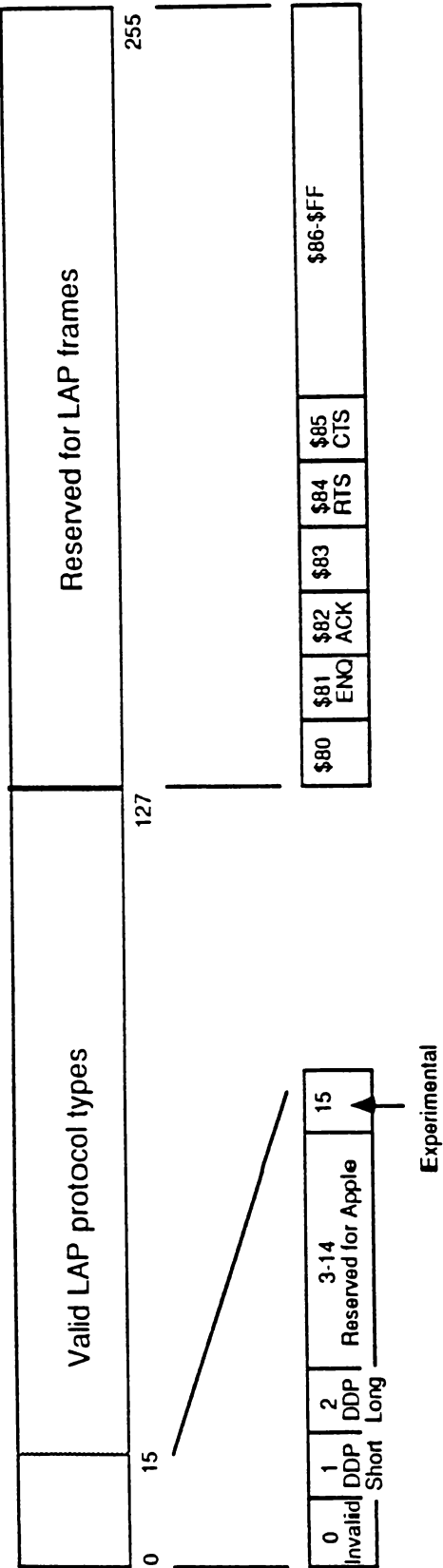
**Figure B-1.  ALAP Protocol type values**

**DDP protocol types**

0 — 15 — 255

| 0 Invalid | 1 RTMP | 2 NBP | 3 ATP | 4 EP | 5 RTMP | 6 ZIP | 7-15 Reserved for Apple |
|---|---|---|---|---|---|---|---|
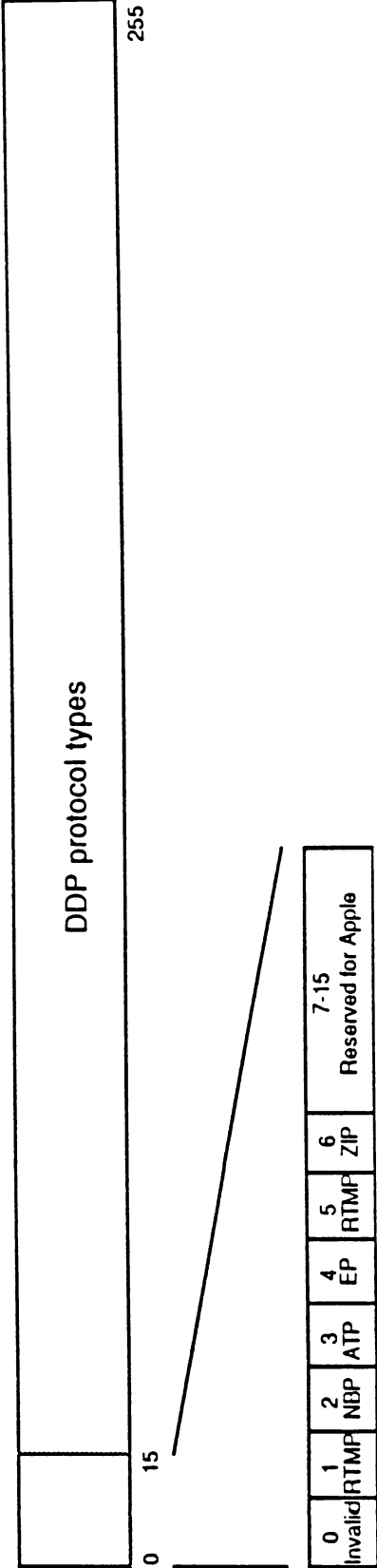
**Figure B-2.  DDP Protocol type values**

Statically Assigned Sockets

Dynamically Assigned Sockets

0    63    127    254

| 0 Invalid | 1 RTMP | 2 NIS | 3 | 4 | 5 | 6 ZIS | 7-63 Reserved for Apple | 64-127 For Experimental Use |

Echoer

Reserved
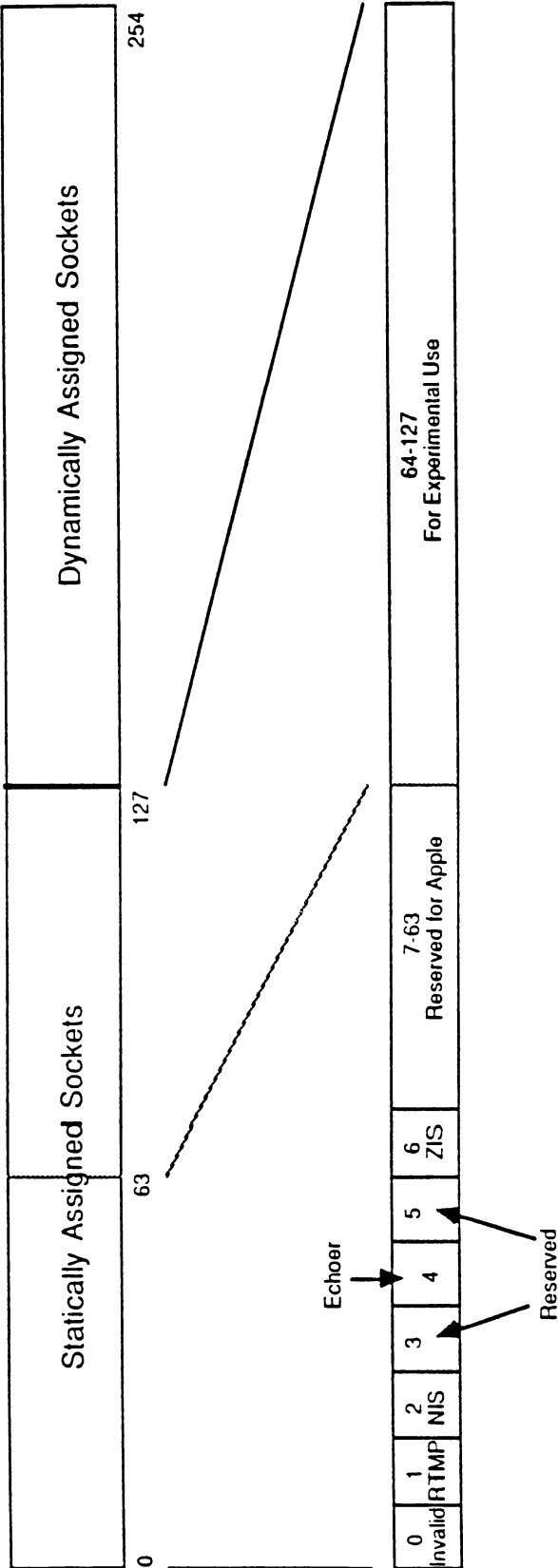
Figure B-3.  DDP Sockets

# Appendix C

# AppleTalk Peek

The *AppleTalk Peek* program is a network tool used to monitor packet traffic on an AppleTalk network. Peek runs on any Macintosh except the Macintosh XL *with AppleTalk connected via the Printer port* , and can record all packets seen on the bus. In addition, it can detect certain errors, measure packet arrival times, and display packet data in hexadecimal and ASCII format.

Peek has enough queue space to hold a large number of packets. This queue is used in a circular fashion, so that Peek can continue to monitor packets even after the queue has been filled. Older packets are discarded to make room for newer ones.

This document describes version 3.0 of Peek.

## The Window

When Peek is started, the program's window is drawn. It contains the control buttons, menus, and information display areas described below.

```
(START)
(STOP)
```

Peek is always in one of two states: recording or displaying packets. When the program is first started, it is in the record state. The **START** and **STOP** buttons are used to initiate and terminate a <u>recording session</u>, during which Peek listens on the bus and records traffic.

When the **START** button is pressed, packet recording is enabled. The button becomes gray to indicate that Peek is recording (see Figure 1). Peek's internal buffers are cleared, and packets from a previous session are lost. The **STOP** button halts recording and causes packets to be displayed, if any were recorded during the session (see Figure 2).

```
┌Pkts in Q┐
│    0    │
└─────────┘
```
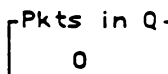
When the **STOP** button is pressed, the "Pkts in Q" box shows the number of packets in Peek's queue. This box is not dynamically updated during a recording session, since queue wraparound makes this determination difficult. The word "Sampling" appears here while recording, as an indication that Peek is monitoring the bus.

The size of the queue is determined by the amount of free memory, so that Peek running on a 512K Macintosh will be able to record more packets than a 128K Macintosh. There is, however, a limitation of 190,000 bytes total in the receive buffer.

```
 ┌─Pkts Rcvd─┐
 │           │
 │     0     │
 └───────────┘
```

The total number of packets seen by Peek since the start of the recording session appears in the "Pkts Rcvd" box. This count is updated dynamically, and hence provides a rudimentary visual indication of bus traffic. Since Peek's queue can wrap around and "forget" old packets, this count may be greater than the number of packets stored in the queue at that time.

```
┌──────────────────────────┐
│ CRC errors:         0     │
│ Overruns:           0     │
│ Time Outs:          0     │
└──────────────────────────┘
```

This box displays the tally of errors during a recording session. Peek can detect three types of errors:

CRC errors are noted when the 16-bit Frame Check Sequence (FCS) at the end of the ALAP frame does not match the calculated FCS. This indicates a possible error in transmission (due to noise on the bus, collisions, etc.).

Overrun errors occur when the receiver reads bytes too slowly from the Macintosh's Serial Communications Controller (SCC), and this chip's three-byte FIFO buffer overflows. Note that if the node running Peek detects an overrun error, it does not necessarily mean that this will be the case for other nodes. This error is sometimes detected as a by-product of collisions on the bus.

Timeout errors are flagged when a byte is expected on the bus (the end of the packet has not been seen, yet a byte does not appear on the bus within about 400 microseconds of the previous byte). Every byte received thus far will be stored and displayed as "the packet". even though the true packet end was not detected. This usually indicates a problem in the packet sender's hardware, but it may also be a by-product of collisions on the bus.

The error fields are updated "on the fly" as packets are recorded, and are a measure of the total number of errors seen by Peek. Therefore, in a long recording session, it is possible. due to queue wraparound, for a packet to be received in error and not appear in the packet display, although the error is noted in the box.

```
┌─Rcv Status──────────────────┐
│ Receiving LAP packets.       │
│ Receiving RTMP packets.      │
└──────────────────────────────┘
```

The "Rcv Status" box indicates to the user the status of two different types of packet reception filters. If a status line is displayed, then that filter is disabled. The default is set to 'filter enabled'.

## Display box

This box is used to view packets which were saved during the recording session. Each packet is preceded by a banner line (see Figure 2) in boldface which includes, from left to right:

- a set of square brackets which may contain blanks or one of the following characters: **B** if the packet was a broadcast, **C, O,** or **T** if a CRC, overrun, or timeout error, respectively, was detected for that packet;
- the source S and destination D node IDs of the packet, in decimal format;
- the packet's arrival time T in milliseconds (measured relative to the first packet stored in the queue);
- the time since the previous packet's arrival (delta time or $\Delta T$);
- the packet's sequence number in parentheses (in the order in which they were received, starting with zero for the oldest packet in the queue);
- the calculated length L of the packet (number of bytes in decimal, not including the FCS).

Following this banner line are two displays of the packet's contents, in hexadecimal on the left and the corresponding ASCII (if printable) on the right. A period is substituted for any unprintable character. Note that the FCS bytes are not shown.

On the right side of the display box is a scroll bar which is enabled whenever there is more to be displayed than will fit in the box. The user can scroll through the display of packets by using the scroll bar's up and down arrows, or by dragging the thumb. Clicking in the gray area above or below the thumb shifts the display backwards or forwards one page at a time. As an alternative, holding down the option key while clicking in the grey area will scroll one entire packet at a time. This is useful for scrolling past large packets.

## Menus

**Menu:**

```
 ⌘  File   Edit   Search   Control
┌─────────────────┐
│ About Peek...   │
├─────────────────┤
│ Control Panel   │
│ Chooser         │
└─────────────────┘
```
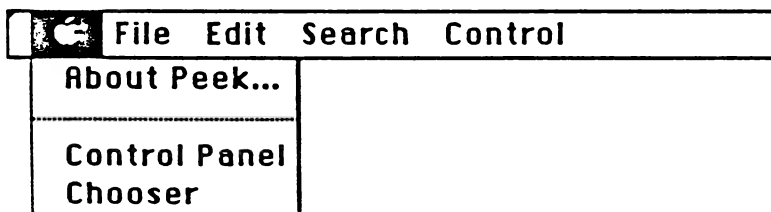
The Apple menu, as usual, is used to invoke a variety of desk accessories. Choosing **About Peek...** will cause Peek to display some descriptive information, including its version number and the size of the queue in bytes.

**File
Menu:**

| ⌘ | **File** | Edit | Search | Control |
|---|---|---|---|---|

**Set Selection Range...**

**Save Selection
Print Selection**

**Quit**          ⌘Q
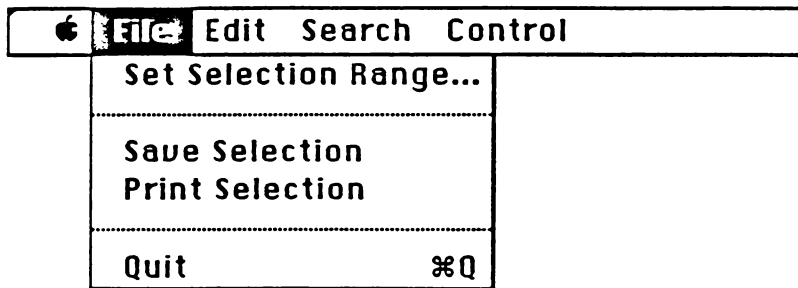
The second menu is the **File** menu, as seen above. Items in this menu allow you to save packet images, print them out, or just exit the application.

Before packets are saved or printed, you may use the **Set Selection Range...** menu item . This command allows you to select a range of packets in the queue to be later saved or printed. The default selection is set to "all" when Peek starts, however you may change this to any range valid for the current buffer. Please note that this selection range is in effect until it is changed or the program restarts.

Selecting **Save Selection** will cause Peek to bring up the standard Save dialog, in which case you may save away a copy of selected packets into a file on disk. If you have used the **Set Selection Range...** command to make a selection range, only those packets will be saved onto your disk.

If there is not enough room on the disk to save all the packets, Peek will write as many as will fit, and notify the user that a "Write Error: -34" occurred. This means that the disk is full; you may wish to put a different disk in the drive and try again. The file containing saved packets will be of type "TEXT" and creator "EDIT".

By choosing **Print Selection**, the selected packet range will be printed onto your printer. As in **Save Selection**, if you have chosen a selection range, only those will be printed. The printer that will be used is the one that was selected using the Choose Printer or Chooser desk accessories.

Note that the **Short Format** option (described later) has no effect on packets written to a file or to the printer; the long format is always used.

C-4

**Edit**
**Menu:**

| ♠ | File | **Edit** | Search | Control |
|---|------|----------|--------|---------|

|       |      |
|-------|------|
| Cut   | ⌘H   |
| Copy  | ⌘C   |
| Paste | ⌘U   |

The **Edit** menu is used only in conjunction with the **Find Pattern** feature in the **Search** menu, described below. It allows you to use the standard text edit commands to create a string for which to search.

**Search**
**Menu:**

| ♠ | File | Edit | **Search** | Control |
|---|------|------|------------|---------|

| Find Pattern | ⌘F |
|--------------|-----|
| Find Same    | ⌘S  |
| ............ |     |
| Find Overrun |     |
| Find CRC Error |   |
| Find Timeout |     |

The **Search** menu is used to look for a particular hexadecimal or ASCII string within the recorded packets. Selecting Find Pattern will cause the Find window to appear. The standard text editing features available in the Edit menu may be used. Hex strings must be a sequence of bytes, each specified as a dollar sign ($) followed by a two-digit hex number. A wild-card may be specified by the command key-equals sign combination. This will appear in the Find window as "Ω", and will match one or more characters of any value.

When the string has been entered, hit Return or the Find Next button. Peek will begin searching from the first packet appearing in the display box. The display will be scrolled down to the packet containing the string, if found, and the string will be highlighted. Otherwise, Peek will inform you that the string was not found. Selecting Find Same (or the equivalent command key-S combination) will cause Peek to look for the next occurrence of the same string, starting from the current packet. Note that searches are always made case-sensitive.

**Find Overrun, Find CRC Error,** and **Find Timeout** work in a similar fashion. Selecting one causes Peek to search, starting from the first packet in the display box, for a packet exhibiting the particular error. If found, the display is scrolled to bring that packet to the top of the box (unless it is too close to the last packet to scroll up to the top). Since the error counts are cumulative from the time the START button was pressed, packets with

errors may not always appear in the queue (if they were discarded to make room for newer packets). If there are no errors of that type, the menu item is dimmed.

## Control
Menu:

```
 É   File   Edit   Search  Control
                          ┌──────────────────────────┐
                          │ Display Short Format Pkts │
                          │ Receive LAP Control Pkts  │
                          │ Receive RTMP Pkts         │
                          └──────────────────────────┘
```
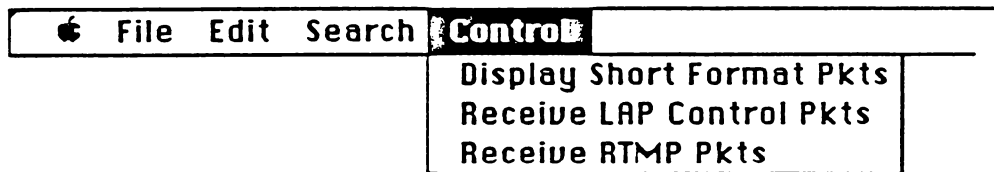
The **Control** menu allows you to manipulate packet information. When **Display Short Format Pkts** is selected, a check mark appears next to the menu item and packets are displayed in a more compact form. Only the banner line and the first line (up to the first 16 bytes) of each packet will be shown. The display can be scrolled as before. Choosing the menu item again will change the display back to long format. This item is inactivated during a recording session.

When **Receive LAP Control Pkts** is selected, all ALAP control packets seen on the bus will be recorded. These are defined as any packets whose ALAP type field is $80 through $FF hex (most significant bit set). Such packets will be recorded only when this option is selected. If their reception is enabled in the middle of a recording session, any ALAP control packets already seen on the bus will not have been saved. Changing this option while viewing the display of a previous session will have no effect on the display, but will affect the next recording session.

The **Receive RTMP Pkts** is used to filter out RTMP type packets. RTMP packets are defined as having a ALAP type field of $01 and a DDP type field of $01. See the appropriate section of Inside AppleTalk for more information on the RTMP protocol.

## Notes

1.  The newer versions of Macsbug (1/1/85 or later, with symbols) tend to slow Peek enough that it will frequently overrun. Use older debuggers on your Peek disk or none at all.

2.  When Peek starts up, it checks low memory variables to see if a RAM routine (like a debugger) is hooked into the trap dispatcher. If one is found, a warning message is displayed, allowing the user to take appropriate action.

3.  A node running Peek is in listen-only mode on an AppleTalk network. Such a node does not participate in the ALAP protocol and does not even consume a node ID. In fact, it is "invisible" to other nodes.

4.  Peek does not use any of the standard AppleTalk drivers (e.g. the Macintosh Protocol Package), but assumes direct control of the Macintosh's AppleTalk port. However, the port is reset when Peek terminates, so it is possible to then run other AppleTalk

software without powering down the Macintosh and powering it up again. (Note that this is not true for versions of Peek older than version 2.0).

4. Peek will not run on a Macintosh XL.

5. If a packet that is longer than 4095 bytes is received by Peek, its subsequent behavior becomes unpredictable. If Peek terminates abnormally during a recording session, there is a strong probability that a node on the network has sent a packet of illegal size (e.g. a node is stuck in its transmit loop).

## Acknowledgements

**⌘ File Edit Search Control**

═══════════════════════════════ **AppleTalk Traffic Peek** ═══════════════════════════════

| START | ┌Pkts in Q┐ | ┌Pkts Rcvd┐ | Overruns: 28 | ┌Rcv Status──────────── |
| STOP | Sampling | 337 | CRC errors: 1 | Receiving LAP packets. |
| | | | Time Outs: 0 | |
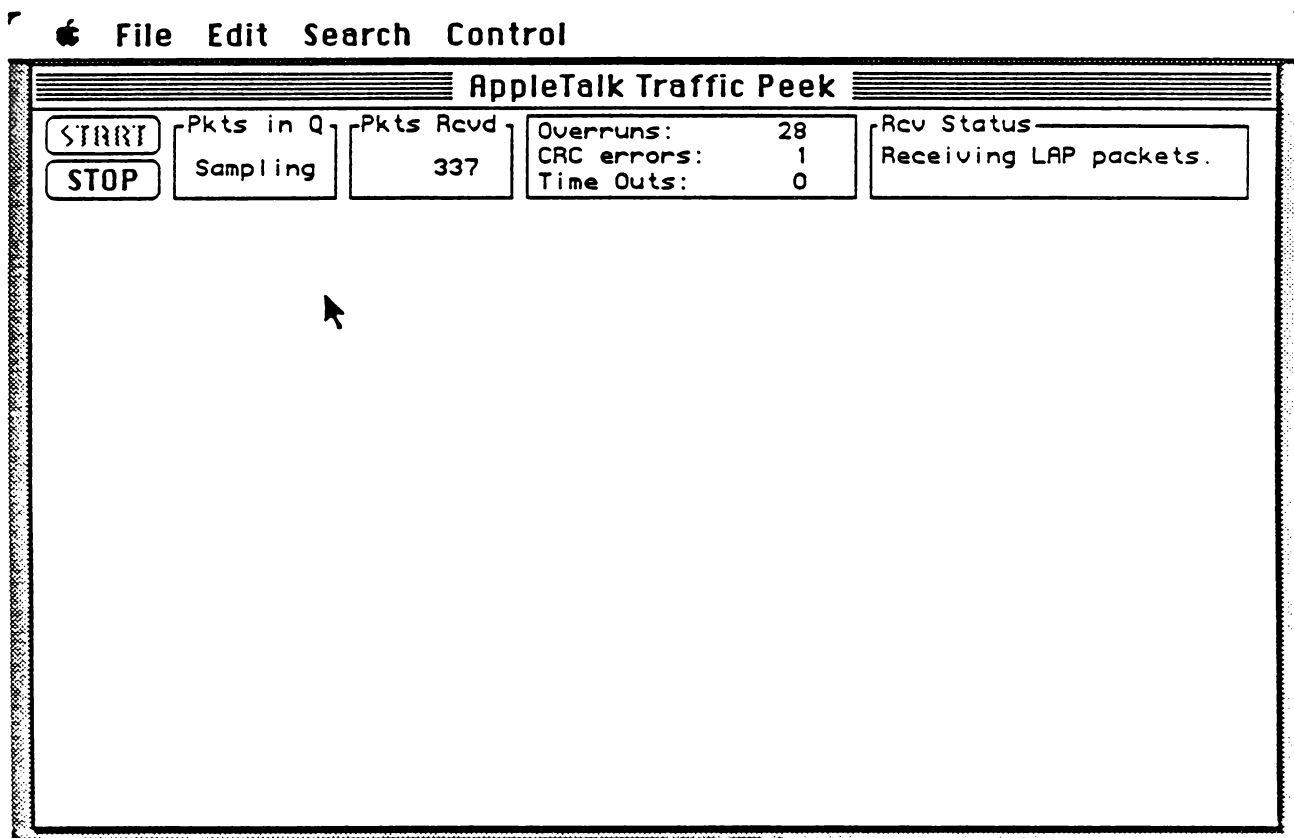
Figure C-1.  Peek Sampling Window

**🍎 File   Edit   Search   Control**

AppleTalk Traffic Peek

| START | ┌Pkts in Q┐ | ┌Pkts Rcvd┐ | Overruns: | 8 | ┌Rcv Status─────── |
|-------|-----------|------------|-----------|---|----------------|
| STOP  | 338       | 338        | CRC errors: | 1 | Receiving LAP packets. |
|       |           |            | Time Outs: | 0 |                |

```
[   ] S: 108 D: 119 T: 33755      ▲T: 3            (0202) L: 3
     77 6C 84                                       wl.
[ C] S: 0   D: 7   T: 33758      ▲T: 3            (0203) L: 2
     07 00
[ O] S: 108 D: 119 T: 33759      ▲T: 1        ▶   (0204) L: 3
     77 6C 13                                       wl.
[   ] S: 108 D: 119 T: 33760      ▲T: 1            (0205) L: 3
     77 6C 84                                       wl.
[   ] S: 119 D: 108 T: 33761      ▲T: 1            (0206) L: 3
     6C 77 85                                       lw.
[   ] S: 108 D: 119 T: 33761      ▲T: 0            (0207) L: 215
     77 6C 02 04 D4 00 00 22 22 11 11 AC 22 C5 DB 03  wl....."" "
     90 01 F8 E8 AE 04 00 00 5F 5F 5F 5F 5F 5F 5F 5F      ._____
     5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F    _____
     5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F    _____
     5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F    _____
```

Figure C-2.  Peek Display Window

# Appendix D

# AppleTalk Poke

**AppleTalk Poke** is a Macintosh application designed for use by AppleTalk developers. It allows the user to edit/create packets and to send them out on AppleTalk. Developers are expected to use Poke to test their protocol software/hardware implementations for AppleTalk products. Poke uses the Macintosh Protocol Package (MPP) for AppleTalk access (details of MPP are discussed elsewhere). This means that the system file of the boot disk must have AppleTalk Installed (e.g. with the Install tool). This has been done to the disk Poke is on.

This document describes the features and use of Poke, version 3.1. It is not intended to instruct the user on the capabilities, features, or specifications of MPP or of the various AppleTalk protocols, nor does it discuss the normal use of the Macintosh's standard editing abilities. (For information on AppleTalk protocols, see the corresponding specification/description document.)

## Startup

After starting Poke, the MPP driver is loaded in (if it isn't currently in memory) and the main window is brought up (Figure D-1). This window displays the Poke station's AppleTalk node ID and packet information. At this stage, the packet information indicates that no packets have been loaded into Poke (packet names are all set to empty). Next to each packet name, there is a pair of buttons labeled **EDIT** and **SEND**. Initially, all **SEND** buttons are dimmed (inactive) because no packets have been loaded. The main window includes an area for displaying any appropriate error or status messages.

The program operates in two different states. When started up, it is in the send state with the main window displayed. When any **EDIT** button is pressed, it goes into an edit state and the packet editing window is displayed (Figure D-2). In this state, the selected packet can be edited. Clicking the edit window's **OK** button will return you back to the main window and the send state.

## Menus and Commands

### 🍎 Menu:

```
┌─────────────────────────────────────────────────┐
│  🍎  File  Edit  Tools                            │
├──────────────────────────┐                       │
│  All About Poke...        │                       │
│  February 13, 1985        │                       │
│  ........................ │                       │
│  Scrapbook                │                       │
│  Alarm Clock              │                       │
│  Note Pad                 │                       │
│  Calculator               │                       │
│  Key Caps                 │                       │
│  Control Panel            │                       │
│  Puzzle                   │                       │
└──────────────────────────┘                       │
```
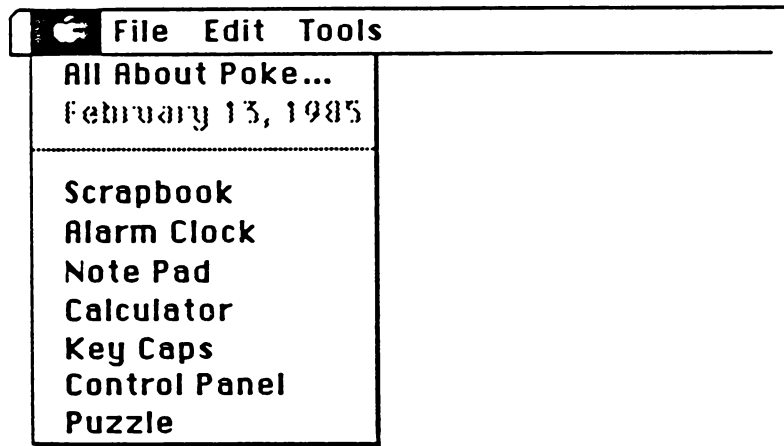
The 🍎 menu allows you to run an available desk accessory or to examine Poke's version information ("All About Poke..."). Selecting the "All About Poke..." command brings up an information window. Clicking the mouse or pressing a key causes this window to disappear and Poke returns to its original state.

### File Menu:

```
┌─────────────────────────────────────────────────┐
│  🍎  File  Edit  Tools                            │
│     ┌──────────────┐                              │
│     │ Load         │                              │
│     │ Save         │                              │
│     │ ............ │                              │
│     │ Quit    ⌘Q   │                              │
│     └──────────────┘                              │
```
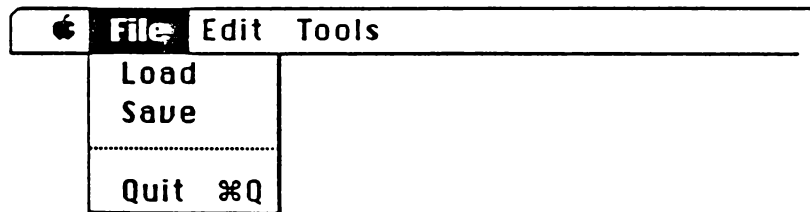
The **File** menu allows the user to load from (or save to) a file of 10 prepared or canned packets. The **Load** and **Save** operations follow the standard conventions for file loading and saving.

> Note: Older versions of Poke utilize a different file format. You cannot load in packets created by those versions.

**Edit** Menu:

```
┌─────────────────────────────────────────────────
│  ⬥  File │ Edit │ Tools
├──────────┬──────────────┐
           │ Cut     ⌘H   │
           │ Copy    ⌘C   │
           │ Paste   ⌘U   │
           └──────────────┘
```
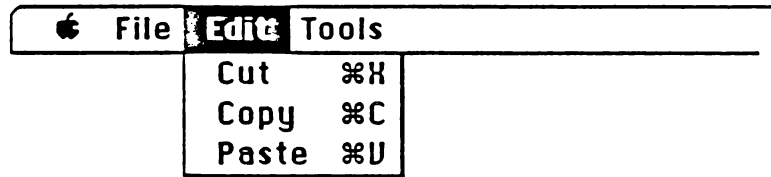
The **Edit** menu is used only while editing a packet. Please note the keyboard's optional command keys that can be used to invoke this menu's commands.

**Tools** Menu:

```
┌─────────────────────────────────────────────────
│  ⬥  File   Edit │ Tools │
├────────────────┬─────────────────────────────────┐
                 │ Clone Packet                     │
                 │ Show Packet Length        ⌘L     │
                 │ Helpful Hints             ⌘H     │
                 │ Set Repeat Factor         ⌘S     │
                 │ Abort Send If Error Occurs       │
                 │·································│
                 │ Calculate CheckSum               │
                 └─────────────────────────────────┘
```
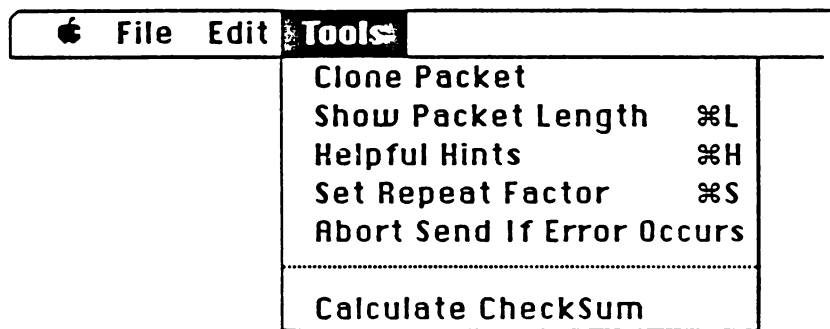
"**Clone Packet**" can only be selected from the main window. When selected, a dialog box appears which asks you for the name of the packet you wish to copy (the source). It also asks for the name of the packet which it is to be copied to (the destination). The names are searched in a top to bottom fashion starting at the top left corner of the main window (Figure D-1). The first packet whose name matches the one you entered will be chosen. If both source and destination names are found, then the source packet will be copied verbatim to the destination. Otherwise, an error message is displayed.

The "**Show Packet Length**" command can only be used while editing a packet. It returns the number of bytes in the packet's data field. This count does not include the packet's header, so the actual packet size will be larger. (See the AppleTalk protocol documentation for information on the size of the different headers.) If an error is detected while computing the length, an alert box will be displayed indicating the exact location of the error.

> Note: If you have entered more data into a packet than is allowed by the corresponding protocol (LAP,DDP,ATP) then Poke will truncate the data (at the end) to the maximum allowed value.

The "**Helpful Hints**" command allows you to obtain a quick summary of editing instructions. Clicking the mouse or pressing any key will return you to the currently active Poke window.

Packets can be transmitted repeatedly at user specified intervals. The number of times a packet is transmitted and the time interval between transmissions are set by the user by selecting the "**Set Repeat Factor**" command. This command will allow you to change transmission information used by the **SEND** command (discussed later).

The delay time interval between transmissions is given in ticks (1 tick = 1/60 of a second). If you enter a number of transmissions value equal to zero, then Poke will keep sending packets out in a closed loop (i.e, indefinitely). When Poke is in such a loop, you can stop the **SEND** operation by either clicking the mouse button or by pressing a key. If you wish to send packets out at the fastest rate possible, enter a zero for the time interval. If this is done, packet statistics will not be displayed in the messages box.

> Note: The user specified time interval is achieved only approximately. Network loading and ALAP overhead plus packet transmit time add to this interval.

The "**Abort Send If Error Occurs**" command is used in conjunction with the **SEND** operation. If selected, a checkmark will appear on the left side of the command informing the user that this feature is active. Now, if an error occurs while sending a packet, the **SEND** operation will abort. To deactivate this feature, select the command again and the checkmark will be removed. This command is especially useful when large numbers of packets are being sent out.

The last command, "**Calculate Checksum**", may be used in the edit window to replace the existing DDP checksum field with an updated checksum. This command is only valid with packets utilizing the DDP long format (LAP Type field $2).

## Preparing a Packet

When you press the edit button for a particular packet in the main window, the edit window of Figure D-2 will appear and you will be shown the information of that packet. This window is divided into two main sections: the header and the data, with 18 editing fields. Only one editing field is active at a time. This is indicated by highlighting that field's rectangular box. There are several circular buttons, check boxes and command buttons (**OK, CANCEL** and **CLEAR**) used in preparing the packet. The standard Macintosh editing features apply to most of these controls. Some, however, need further clarification. These are:

> Pressing the **TAB** key causes Poke to verify the information in the current field before activating the next field. The same is true if you press the **RETURN** key (except within the packet's data field). If an error is detected while verifying a field, a beep will sound and Poke will return you back to the error's location. (Possible errors are described at the end of this section.)

> • Clicking the mouse on a different editing field will verify the information in the currently active field. If there are no errors, Poke moves to the field clicked on.

> You may type data beyond that visible in the field. Leading blanks are automatically removed in the packet header fields.

## Entering the Packet's Name

The packet's name is used only to visually distinguish the various packets from others in the main window. It may contain any sequence of printable characters, but it is suggested that you limit the number of characters to 16.

## Entering Information in the Header Fields

Information in the packet header fields can be entered in any one of three ways:

Decimal :        Type in the digits (e.g. 128). This is the default entry type.
Hexadecimal :    All hexadecimal (hex) numbers are preceeded by a dollar sign
                 (e.g., $80 = 128).
Binary:          Binary numbers are preceeded by a percent sign (e.g., %1111 =
                 $0F = 15).

Leading zeros are ignored. When a field has been verified, the number entered is automatically converted to hex format.

## Possible Error Conditions

Value in field is out of range. (see AppleTalk Protocol documents for the permissible ranges of the various fields)
Unknown character in field. Valid digits for decimal format are [0..9] (where this represents a range from zero to nine); valid digits for hex format is [0..9, a..f, A..F], and valid digits for binary numbers are [0,1].

## Entering Packet Data Information

The following format must be followed when entering information into the packet data.

Data bytes can be entered into the packet in two ways: by typing in the ASCII character corresponding to the byte's value or by entering the byte's value in its hex form.

To enter the hex form, type a "$" followed by the two digit hex number (e.g. $84,$01). Note that "$1" is invalid, you must enter "$01". Byte's whose value corresponds in the ASCII code to a graphic character can be entered by just typing in that character. Example: to enter a byte with the value "$62", type "b"; for "$42" type "B"; for "$31" type "1" Other examples can be found in Figures D-2 and D-3.

> Note: Since the dollar sign ($) is a special character, you can only enter it in its hex form "$24".

Poke will detect errors from the end of the data back to its beginning.

## Editing Buttons

Various buttons in the edit window control the information that constitutes the packet. Each set of buttons is described below:

Packet Type: ○ LAP     ○ DDP     ◉ ATP

The Packet Type buttons are used to choose the header type as described in the protocols document. After clicking on a button, only the fields appropriate for that protocol type will be shown. The default is **ATP**. Only one button may be selected at a time.

○ Req  ◉ Rsp  ○ Rel

These three buttons are only used for an ATP packet. They are used to format an ATP request, response or release packet. The default is **Req**. As above, only one button may be chosen at a time.

☐ XO  ☐ EOM  ☐ STS

Each of these check boxes represents the corresponding bit in the ATP control field. If checked, the corresponding ATP control field bit will be set; otherwise the bit is cleared.

┌─ Packet Data Display ─────┐
│   ○ Hex     ◉ ASCII   │
└───────────────────────────┘

The Packet Data Display buttons allow the user to select the type of display for the packet's data: hex strings or mixed ASCII and hex.

Note: This operation may take up to 10 seconds for large packets.

If an error occurs during the format conversion, an error message is displayed and the conversion will abort. You may enter data in either format at any time. The above buttons are used only when the display is updated or when you wish to convert data to the format immediately.

┌─────── ┐   ┌─────────┐   ┌─────────┐
│   OK   │   │ CANCEL  │   │  CLEAR  │
└────────┘   └─────────┘   └─────────┘

The **OK** button should be pressed when you are through editing the packet. All fields are verified for correctness and the packet length is displayed before returning to the main window. You will also have the option, at this time, to calculate a checksum for the packet. If any errors are detected, you will be returned to the edit window.

The **CANCEL** button terminates the editing session without saving any changes to the packet. The packet is returned to the original form that it had prior to this editing attempt. Poke returns you to the send window.

The **CLEAR** button clears all editing fields and inserts the default information into them.


## Sending Packets

To send packets, Poke must be in the send state (i.e., displaying the main window). Any one of the ten packets may be sent by clicking on its active **SEND** button. The number of times the packet will be sent and the delay between each of these transmissions is shown at the top right corner of the main window in the short form:

$$\text{Rpt Factor} = nx : d \text{ ticks}$$

where:    **n** = number of transmissions
         **d** = time interval between transmissions (in ticks)

If a **SEND** button is inactive, you must first edit the packet. The result of the **SEND** operation is displayed in the message area at the bottom of the main window.


<u>Possible Error Conditions</u>

• No error; packet was sent to destination node (or broadcast)
• -95; Packet was unable to be sent because either the destination node did not respond or the line was sensed "in use" 32 times.


<u>Startup Notes</u>

When Poke starts up, the MPP driver is opened and initialized. If the open call fails and you are returned a -35 error, you will be forced to exit the program. Most likely the cause of this error will be that the MPP driver is not installed in the System resource file. In addition, if the system heap is fragmented such that the MPP driver cannot get enough memory to load, the same error will be returned.

If the serial port configuration byte (SPConfig) is not set correctly, you will get a -98 error when Poke starts. See the AppleTalk Manager manual for additional information on location and contents of this byte.


## Caveats

• Editing of the packet's data field will slow down appreciably as its size increases. Whenever possible, display it under the ASCII mode to minimize the number of screen characters.
• While in the ASCII display, all characters in the printable ASCII range ($20-$7E) and **RETURN** ($0D) will be displayed in their ASCII form, even if they were entered as hex strings.

- The packet data field is limited to 55 lines. Even short packets (e.g., entering more than 55 carriage returns in the packet's data field in ASCII mode) can go out of the scrolling range.
- Numbers cannot be entered into the packet's data field in decimal or binary format.
- In no case can the size of the packet be greater than 603 bytes, including ALAP header.
- If an error occurs while verifying or converting the packet's data field, the information at the error location may change, as Poke tries to back out of the error gracefully.
- If you have chosen DDP or ATP packet types from the edit window, DDP long format will always be displayed, even if the ALAP type of $01 (short format) was entered.
- If you enter more than 600 bytes of packet data, the checksum calculation may not work correctly until you have exited and reentered the editing window. (This will truncate all excess data from the end of the packet).
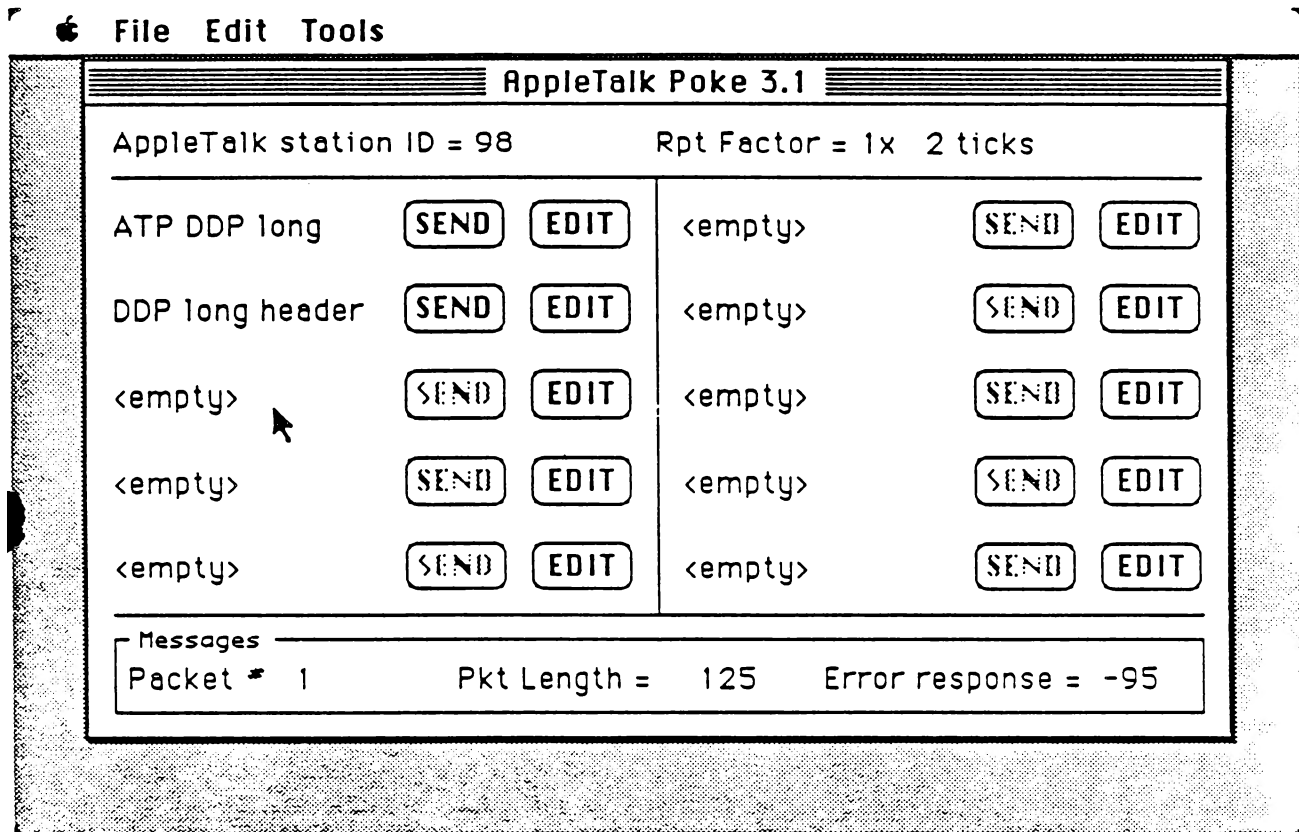
**Figure D-1. Poke Main Window**

Packet Name: [ATP DDP long]        Packet Type:  ○ LAP     ○ DDP     ◉ ATP

┌─ Header Data ─────────────────────────────────────────────────────────────┐
│ LAP ·······································································     │
│        Dest Node Addr: [$45]           LAP Type: [$2]                       │
│                                                                             │
│ DDP ·······································································     │
│        Hop Count:   [$0]      Dest Skt #:       [$A]      Src Skt #:   [$B]  │
│        DDP Type:    [$3]      Dest Node Addr:  [$FF]      Src Node Addr: [$17]│
│        Checksum:  [$1202]     Dest Net #:   [$1]          Src Net #:   [$2]  │
│                                                                             │
│ ATP ·······································································     │
│      ◉ Req  ○ Rsp  ○ Rel    Trans ID: [$FFF]        ⊠ XO  ☐ EOM  ☐ STS      │
│      BitMap:      [$0            ]         U1: [$1] U2: [$2] U3: [$3] U4: [$4]│
└─────────────────────────────────────────────────────────────────────────────┘

┌─ Packet Data ───────────────────────────────────────────────────────────┐⇧│
│ This is ASCII data.  If I wish to enter unprintable characters, I enter    │ │
│ characters like: $00,$01.                                                  │ │
│                                                                            │ │
│                                                                           ⇩│
└───────────────────────────────────────────────────────────────────────────┘

┌─ Packet Data Display ──────┐
│  ○ Hex      ◉ ASCII        │       [  OK  ]   [ CANCEL ]   [ CLEAR ]
└────────────────────────────┘

Figure D-2.  Poke Edit Window